UNIFRAME

A Framework for Seamless Interoperation of Heterogeneous Distributed Software Components

(Final Report)

20050705 053

Funded by the DoD and Office of Naval Research
under the CIP/SW Program

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 06/27/05	2. REPORT TYPE Final Technical Report	3. DATES COVERED *(From - To)* 05/01/01-3/31/05

4. TITLE AND SUBTITLE A Framework for Seamless Interoperation of Heterogeneous Distributed Software components	5a. CONTRACT NUMBER N00014-01-1-0746
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Raje, Rajeev, R., Olson, Andrew, M., Bryant, Barrett, R., Burt, Carol, C., and Auguston, Mikhail	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Indiana University Research and Sponsored Programs 620 Union Drive, Room 618 Indianapolis, IN 46202-5167	8. PERFORMING ORGANIZATION REPORT NUMBER TR-CIS-0624-05
9. ONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) . Ralph Wachter Office of Naval Research, ONR311 Ballston Center Tower One 800 N. Quincy Street, Arlington, VA 22217-5660	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Public Availability (UU)

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The UniFrame research was supported under the CIP/SW Program. The vision of this research is to automate the process of integrating heterogeneous and distributed systems that conform to specific quality requirements. This research addressed three key challenges : a) architecture-based interoperability, b) distributed resource discovery, and c) validation of quality requirements. Principles and prototypical systems were created to demonstrate the successful completion of the research.

15. SUBJECT TERMS
Component-based software, distributed computing, Quality of Service, Model-driven architecture.

. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Rajeev R. Raje
U	U	U	UU	370	19b. TELEPHONE NUMBER *(Include area code)* 317-274-5174

Participants

Rajeev R. Raje
Andrew M. Olson

Barrett R. Bryant
Carol C. Burt

Mikhail Auguston

http://www.cs.iupui.edu/uniFrame

Final Report

Contract Information

Contract Number	N00014-01-1-0746
Title of Research	A Framework for Seamless Interoperation of Heterogeneous Distributed Software Components
Principal Investigator	Rajeev R. Raje
Organization	Indiana University Purdue University Indianapolis

Technical Section

Executive Summary

This is the final report for the project entitled "A Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", supported by the Office of Naval Research under the CIP/SW program, and which was jointly investigated by the Indiana University Purdue University Indianapolis, The University of Alabama at Birmingham, and New Mexico State University/Naval Postgraduate School. The objectives, approach, results achieved, publications, presentations, interactions with other organizations, and educational impact are described in this report. Copies of sample publications are also included as an appendix.

The vision of this research is to automate the process of integrating heterogeneous and distributed software components, so as to create distributed systems that conform to specific quality requirements. The research addressed three key issues while creating a framework, called UniFrame, as a first step towards achieving the vision of the research. These three issues were: a) architecture-based interoperability, b) distributed resource discovery, and c) validation of the quality requirements. The underlying principles for the research are based on model-driven generation, multi-level specification of components, a proactive and distributed discovery of resources, and formalism based on Two-level Grammars and Event Grammars. Many different prototypes were created as a proof of concept for this research and were empirically validated. The results were extensively published in professional forums, such as journals, conferences, and invited presentations. The results of the research were also incorporated in the curricula at all the participating organizations and benefited undergraduate as well as graduate students. The infrastructures at the participating universities were also enhanced by creating dedicated laboratories for the research. The results achieved during the life of the project have successfully demonstrated the soundness of the UniFrame principles, thereby, reaffirming the belief that UniFrame presents a comprehensive approach for constructing distributed component-based systems that satisfy quality requirements.

Objectives

The vision of this research is to automate the process of integrating heterogeneous components to create distributed software systems that conform to quality requirements. As a first step towards accomplishing this ambitious vision, the objective of this research contract was to create a comprehensive framework, called UniFrame, that will enable a seamless interoperation of heterogeneous distributed components.

The key research issues, investigated by this project, to achieve the above mentioned objective are:

i) Architecture-based interoperability
ii) Distributed resource discovery
iii) Validation of quality requirements

Each of these research issues requires addressing several challenging topics, which are indicated below.

i) Architecture-based interoperability
   a. To investigate the roles of modeling, mapping, and automation in achieving interoperability between heterogeneous components.
   b. To develop principles for designing the necessary tools and implement proof-of-concept prototypes.
   c. To explore the challenges related to model-based standardization in the domain of heterogeneous distributed components.

ii) Distributed resource discovery
   a. To develop a multi-level specification mechanism, based on a meta-model that utilizes and facilitates the principle of design by multi-level contract.
   b. To create an infrastructure for publication and distribution of the software components.
   c. To design and experiment with a prototypical system that will provide the hierarchical discovery and selection mechanisms for locating appropriate distributed components.

iii) Validation of quality requirements
   a. To develop a comprehensive vocabulary and associated metrics for the quality parameters of a system.
   b. To investigate the composition and decomposition rules for these quality parameters.
   c. To create the necessary formalism for monitoring events related to these quality parameters.

Approach

The assumptions of the research are: a) distributed system construction is to be achieved by integrating independently deployed heterogeneous components, and b) automation aids in increasing the quality of the generated system and requires less development time. The technical approach used in UniFrame is based on principles of: a) model-driven design and construction, b) distributed discovery using multi-level specifications, and c) Two-Level and event grammars.

Figure 1 indicates the UniFrame process [BC-1, CW-1][1] for constructing distributed systems from geographically distributed heterogeneous software components. The central piece in the UniFrame process is a comprehensive knowledgebase (KB) that contains detailed descriptions of a) a service-based architecture (modeled by feature diagrams) for a family of systems for the particular type of application under consideration, b) rules for matching and selecting distributed components, c) rules for semi-automatically generating a distributed system from selected components, and d) the rules for the description, the instrumentation and the measurement of the quality requirements of the generated system. In the current research effort, it is assumed that this KB is developed by domain experts, such as various task forces of the Object Management Group (OMG), using the available standards. However, for the purposes of the prototypical development and experiment, this KB has been handcrafted. Details of the prototype KB are provided in [DT-4], whereas the architecture and application of the more general concept are explained in [BC-1, CW-1].

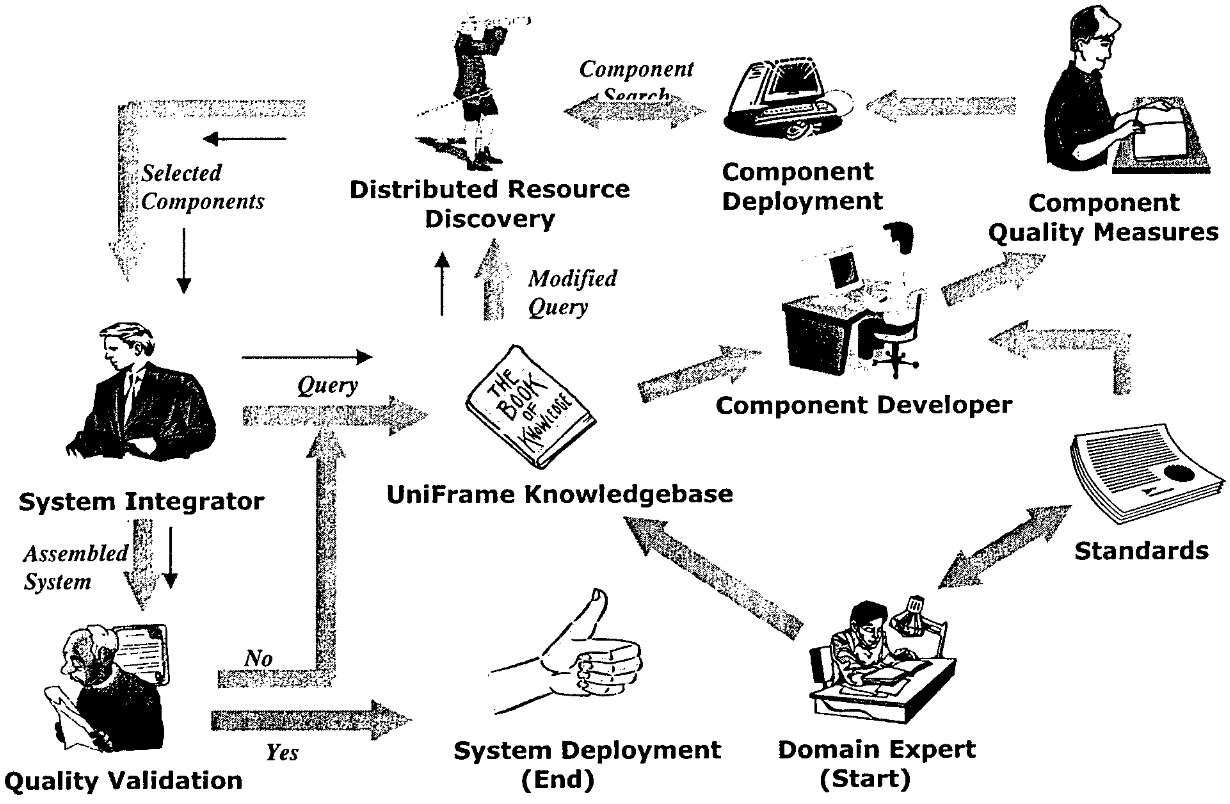


Figure 1: The UniFrame Approach

In the UniFrame process, it is assumed that developers independently create components using a specific distributed component technology that adhere to the KB standards. In addition to creating, validating, and deploying the components, they are required to develop a multi-level specification, called a UMM (Unified Meta-component Model) specification [OR-1, CW-1], for each of their components. This UMM specification is an enhanced version of the multi-level contract principle advocated in [OR-2]. Each component has four levels of contract: a) syntax, b) semantics, c) synchronization, and d) Quality of

[1] The following scheme is used in indicating citations. These references are divided into: a) Book Chapters (BC), b) Journal Papers (JP), c) Conference and Workshop Proceedings (CW), d) Standards Documents (SD), e) Dissertation and Theses (DT), and f) Other References (OR). Each citation contains a prefix, which indicates the corresponding category (e.g., BC), followed by a number (e.g., BC-1).

Service (QoS). Various mechanisms are utilized to describe these different levels of the contract. For example, the semantics contract uses pre-, post-conditions and invariants, while the QoS level contract indicates the appropriate QoS parameters with their metrics and their behavior as a function of the execution environment. Examples of UMM specifications are available in [BC-1, CW-1]. The component along with it's UMM specification (in XML) is deployed on the network using the underlying infrastructure provided by the technology that was used to develop the component. Once a component is deployed, it is available for discovery.

The task of locating a component is carried out by the UniFrame Resource Discovery Service (URDS). URDS is hierarchical, proactive, interoperable, and decentralized in nature. Its three constituents are: a) active registries (AR), b) headhunters (HH), and c) Internet component broker (ICB). ARs are the native registration mechanisms of different distributed component technologies (such as the registry in Java-RMI) except for the fact that they are proactive in nature, i.e., they are always listening for communications from HHs. HHs are responsible for discovering components that are deployed on a network. Once a HH discovers components, it registers them in its local store, called meta-repository (MR). HHs are responsible for matching requests for components with the available components and also propagating these requests to other HHs. The ICB is a collection of various services such as the authentication, federation, and query processing. Comprehensive details of URDS are available in [DT-1, CW-15].

The discovery process is initiated by a request, or query, for a distributed system from a system integrator with the intent of constructing it from components deployed on a network. This query indicates the nature and the features of the desired distributed system. The features include a combination of a variety of QoS parameters (such as turn-around time < 200 ms) and a type of the desired system. The query manager (a part of the ICB) uses the KB to determine a system design instance out of the families of systems stored in the KB that is appropriate for the query. Once that instance is identified, the query is decomposed into sub-queries, each indicating specific types of components, along with their QoS features, that are needed to construct the desired distributed system. This decomposition process uses the rules that are described in the KB. The details of this process are described in [DT-4, DT-5, CW-23]. These sub-queries are supplied to the URDS for locating the components that match their criteria appropriately. Once components are located, they are presented back to the system integrator for selecting, in case there is more than one candidate for a given sub-query.

Once the system integrator selects a complete set of components, the system generator described in [DT-4, CW-24] constructs a distributed system from them. The construction process does utilize the generation rules, which are described in the KB, that express the architecture of the system design. In addition, the construction process instruments the necessary QoS-related code into the integrated system. The generation process uses the principles of Two-level Grammar (TLG) [CW-16], and the instrumentation process is based on the concepts of event grammars [OR-3]. Once the system is integrated, composition rules, which are part of the KB, are used to make a prediction about the QoS of the entire system based on the individual QoS values. Then, the prediction is validated against the actual values obtained from collecting the event traces (based on the event grammars) resulting from experimental testing of the system. The details of composition rules are in [DT-5].

It is possible to perform the discovery, generation, integration and validation in an iterative manner. For example, the discovery process may yield no satisfactory components, or the integrated system may not meet the desired QoS requirements. In such scenarios, the query could be reissued, with possible modifications, and reprocessed by the URDS. Such an iterative process provides the necessary flexibility of an incremental design. This is discussed further in [BC-1, BC-3].

The salient features and scientific merits of the UniFrame approach are that it:

i) provides a unified approach through the UniFrame Process (described briefly above).
ii) uses the principles of model-driven (KB-based) automation for the system construction.
iii) allows interoperation among heterogeneous software components meeting QoS requirements.
iv) uses a meta-model-based approach for multi-level specification of components.
v) follows a proactive advertisement and discovery of components.
vi) proposes a Quality of Service Framework that contains a QoS catalog and a unifying system monitoring technique.

The UniFrame research offers the following benefits to the CIP/SW initiative:

i) The UniFrame process, supported by appropriate tools, will enable a semi-automatic distributed system assembly from heterogeneous and distributed software components.
ii) The standards-based technology will be enhanced to achieve a seamless integration of the heterogeneous components.
iii) A semi-automated system assembly with integrated validation metrics will improve the system quality.

The UniFrame research project has supported students at all levels, from the undergraduate to MS and PhD, at the participating institutes, thereby, enriching their educational experience. In addition, various research topics that are being investigated have been incorporated into the curricula at all the participating institutions. Specialized computing laboratories have been created as a result of the UniFrame research.

Accomplishments

A brief summary of the tasks accomplished during the life of the UniFrame research (2001-2005) is presented below. It is classified under the three key challenges that UniFrame research is addressing. Detailed technical aspects of the accomplishments are published in many papers, a listing of which is provided under the publications section of the report and a few representative papers are attached as an appendix.

i) Architectural-based interoperability

The challenge of architecture-based interoperability was tackled using a multi-pronged approach. First, a formal process, based on the UniFrame principles, was designed [BC-1, BC-3, DT-2]. Second, the contents and the formalization of the KB were carried out [DT-4, DT-5, DT-24, CW-33], and associated model-transformation techniques were explored [JP-6, CW-43]. Third, principles, based on the applicability of the Two-level grammar, for the generation of the glue code were identified [JP-4, CW-13, CW-16, CW-17, CW-22, CW-25, CW-29, CW-40, CW-41, CW-44, CW-45, DT-23, DT-24], and prototypes were created which generated glues and hence, inter-operable distributed systems [DT-4, DT-10, DT-23, DT-24, CW-30]. Finally, preliminary explorations about the applicability of the UniFrame approach to other domains, such as Grid Computing, were carried out [DT-21, JP-7, CW-46]. The significant results achieved while addressing this challenge are described below:

a. The activity of formulating the requirements for a family of computing systems in a particular domain was studied as part of the effort to develop the knowledgebase. A formal, machine processable language was developed for expressing the domain's alternatives in terms of feature graphs [DT-2]. This work was extended to representations of designs of systems in the domain [DT-4, CW-24], A process for developing component-based systems within the UniFrame context was formalized [BC-1, BC-3], and a prototype was developed [DT-4]. An example problem was examined to illustrate the OMG's Model Driven Architecture (MDA) approach to generating platform specific designs from platform independent models [CW-19]. With this prototype, a system developer could formulate the requirements for a specific system in the domain and generate an appropriate, component-based implementation automatically.

b. Because the effort of creating the knowledgebase is so extensive, the domain experts with this task must have at their disposal adequate representation languages and strong support tools. A major problem is that feature graphs are an incomplete representation of the systems within a domain because they represent essentially just alternatives among the systems. A more complete design representation requires a standard system design language, such as UML, with an extension for representing the feature variants that typify a design family. To address this issue, a study was made of a number of such extensions. However, none was found that is mature enough to have tools with which to generate the domain representations in an open form satisfactory for use by the UniFrame development process described above. The closest product found is proprietary. It is based on the concept of 'archetype patterns', a generalization of design patterns. It permits the UML-like abstract representation of alternative designs in a domain, and can generate code in an MDA fashion, given what is called a "cartridge" for the appropriate language. Existing cartridges do not permit construction at the component-based level. This survey and analysis, with example applications, are reported in [DT-16].

c. The integration of the aspect-orientation into generative domain modeling for modeling component domains was completed. This approach to modeling facilitates component specification as well as component integration. The approach to modeling Web Services (WS) for the purpose of integrating components following the web services technology domain model was specifically investigated [JP-5, CW-35, CW-41, DT-23]. Software systems can incorporate WS technology in order to be reused and integrated in a distributed environment across heterogeneous platforms. The following issues were specifically addressed: 1) the migration of legacy distributed software systems toward WS applications; 2) the innovation of new infrastructure, and languages in support of WS application development. The relationships between this type of model and traditional Entity-Relationship diagrams was also explored [CW-47].

d. The hypothesis that the UniFrame model of components distributed over the network was an embodiment of the semantic web was further investigated [BC-2]. UniFrame exists in a semantic web of software components. The natural language foundation of UniFrame queries may also be used in querying the semantic web [JP-2, DT-7, CW-6, CW-8, CW-9, CW-18, CW-26, CW-38].

e. Two-Level Grammar (TLG) [CW-5, CW-39] continued to be applied to model the feature composition in domain models [JP-8, CW-14, CW-37, CW-43]. The foundation of grammar for this approach appears to offer a number of interesting possibilities for model-driven development, as detailed in [CW-12, CW-21, CW-28].

f. As the UniFrame process places a strong emphasis on the incorporation of the QoS parameters during the entire life cycle of distributed component-based systems, it was necessary to develop a mechanism that allowed the depiction of the QoS parameters during the design stage. For that purpose, the concept of the collaboration diagram was extended to incorporate the modeling of QoS parameters during the design phase. Case studies were carried out to assess the effectiveness of this approach [DT-12]. It demonstrated that annotated collaboration diagrams are an effective mechanism for modeling the QoS parameters during the design of distributed systems. Formal specification methods for QoS were also investigated [JP-3, CW-11, CW-36].

g. As indicated earlier, heterogeneity is the main challenge that UniFrame has to address; a template-based approach was investigated and a prototype was created that allowed an interoperation between Java-RMI and CORBA components. The effects of different placements of the generated glue on the performance metrics (such as the turn-around time) were observed by experimenting with the prototype [DT-10]. Different alternatives (such as centralized or one-to-one distributed) were selected for the placement of the glue that is generated using the template approach. It was observed that the distributed placement on the initiator and responder machines yielded less performance penalty and thus, is a better choice than the other alternatives, such as the centralized placement.

h. A preliminary exploration about the applicability of UniFrame principles to other domains was initiated by considering grid computing as a potential target. Grid computing, due to its inherent distributed, heterogeneous and quality-aware (e.g., precision, speedup) properties, represents an ideal target domain. The simplified structure of the KB for a grid-related application was presented in [DT-21 CW-46, JP-7] and a few experiments that allowed an interoperability of UniFrame components with Grid-based components (developed using Globus) were designed and executed. These preliminary investigations indicate a potential for UniFrame to act as a formal process for grid-related application development.

ii) Distributed resource discovery

One of the assumptions in the UniFrame research is that a distributed system is realized by integrating various heterogeneous and independently created components. Thus, the process of discovering components is not only a pre-requisite for generating a distributed system, but also a critical task in the UniFrame process. This task of discovery in UniFrame, as indicated earlier, is performed by the UniFrame Resource Discovery System (URDS). Many different aspects of URDS were investigated in the research. The highlights of this exploration are:

a. The starting point for the URDS was the design of its architecture. [DT-1, CW-15] This architecture is hierarchical, proactive, and allows an interoperation across different component models. An initial prototype was created in [DT-1] using the J2EE and was experimented with. The results established the validation of the architecture and the proactive nature of URDS.

b. As a next step, the applicability of the URDS architecture to the .Net component model was investigated in [DT-6, DT-13]. The prototypes developed in this effort used the

UDDI mechanism of the .Net model. [DT-13, CW-31] explored an extensive comparison between the .Net model and the UniFrame paradigm in general, with a specific focus on URDS. The principles of URDS were found applicable in the context of the .Net model as well and this prototype was made to interoperate with the one developed using J2EE model, thereby, indicating the interoperable nature of the URDS architecture.

c. Investigations indicated in (a) and (b) were carried out on prototypes which were moderately sized. Thus, to explore the scalability of the URDS, a simulation was created [DT-21]. This allowed the investigation of different configurations of URDS constituents and their impact on the performance, measured by the time required for serving a request. These experiments indicated that the URDS architecture is scalable, which is attributed to its hierarchical nature.

d. In addition to empirically evaluating the scalability of the URDS architecture, investigations were carried out to perform selective search techniques while locating appropriate components. This required designing different query propagation schemes and studying their effect on the performance (e.g., response time) and the quality of the components discovered (e.g., precision). Different query propagations were carried out using the concept of acquaintances [OR-4], which uses the principles of reinforcement learning [OR-5]. A Headhunter, in addition to searching its local meta-repository, propagates the incoming query to other headhunters. Thus, the search process is equivalent to traversing different graphs created due to the selection of acquaintances. Different techniques such as, random, short-term, long-term, and profile-based, were designed and used in deciding the acquaintances [DT-19, DT-21, JP-10]. Experiments were carried out to study the impact of these techniques on different query and component distributions. It was shown that the long-term and profile-based techniques performed better in most of the cases.

e. The above mentioned investigations used simple techniques, based on component types and QoS values, for matching queries with the available component specifications. Such a matching, although simple, is far from comprehensive. Also, it does not take advantage of the other levels of the contracts (e.g., semantics, and synchronization) that a UMM specification employs. Hence, investigations were carried out for multi-level matching during the discovery process. In [OR-6, OR-7] techniques are described that allow the matching at the syntax and semantics levels. These are based on the type relations and predicate logic. These principles acted as the starting point for the investigations of multi-level matching. These explorations resulted in identifying the formal structure of the synchronization and QoS contracts and associated matching principles [DT-15, JP-9]. Rules for different types of matching (e.g., exact-match and relaxed-match) were created and validated by developing a synchronization policy catalog and using the temporal logic of actions [OR-8]. Different operators were also defined for matching the QoS level contracts. Such a multi-level matching is more comprehensive than the simple matching that URDS prototypes were employing. An incorporation of the multi-level matching is future work that is being currently investigated.

f. During the empirical evaluation of the scalability of URDS, it was realized that the monitoring and management functions were largely done in a manual manner. Thus, a GUI-based monitoring and management system for URDS was created [DT-20]. This system uses the model-view-controller pattern and employs the principles of event-driven as well as periodic modeling. It provides two views, one for the manager of the URDS and the other for the user of the URDS. Empirical evaluations were carried out to demonstrate the effectiveness of this system.

g. The incorporation of mobility into the URDS architecture was carefully evaluated and a design that encompasses mobile headhunters was created. A mobile agent-based version of URDS (called MURDS) was created and experimented with to show the effectiveness of the incorporation of the mobility into the URDS architecture [DT-11].

h. The multi-level contracts make the task of creating the UMM specification for a component fairly elaborate and increase the complexity during the component development process. Hence, a UMM-specification editor was developed that assists in this task [DT-9].

iii) **Validation of quality requirements**

As indicated earlier, UniFrame emphasizes the quality requirements throughout the development of the distributed systems. It specially focuses on the QoS features, as these features are critical in many different application domains, such as distributed real-time systems. The approach followed to address this challenge was: a) to create a vocabulary of the QoS parameters and to study the effects of the environment on the QoS parameters, b) to propose a methodology for incorporating QoS parameters in a model-driven approach, c) to propose a methodology and associated tools for empirically validating the QoS parameters, d) to study access control as a QoS parameter, and e) to propose and investigate the applicability of UniFrame's QoS approach to distributed real-time systems. These are briefly discussed below.

a. As a first step towards addressing the QoS aspects, a comprehensive QoS catalog was created [CW-4]. This catalog contains commonly used QoS parameters, along with a description of their features (such as intent, method of evaluation, etc.). The structure of the catalog is loosely based on the structure of the design patterns catalog [OR-9]. Thus, each parameter, in this catalog, is described using a template that is similar to the one used in the design patterns catalog. Models for measuring the parameters are also indicated in the catalog and have been experimentally validated for the dynamic parameters. These parameters are classified based on their nature (e.g., static/dynamic) and on the application domains that they appear in. This was further enhanced to create a QoS-based framework for UniFrame [JP-1]. Also, the effects of different factors, such as the execution environment and usage patterns, on dynamic QoS parameters were also studied in [DT-3]. This work also acted as a foundation for the QoS composition and decomposition rules that were studied in [DT-5]. Composition and decomposition rules for parameters from the catalog were created and empirically validated [DT-5, CW-23].

b. The incorporation of QoS framework (as described in (a)), into OMG's model driven architecture (MDA) was investigated [CW-7]. Based on this, an approach for transforming QoS features from the platform independent models to platform specific models was developed [CW-19]. This work was also discussed with different task forces of the OMG and also resulted in the corresponding OMG RFPs [SD-1, SD-2].

c. The investigations in (b) were further expanded by specifically focusing on access control as a QoS parameter. In [CW-32] an approach was presented to unify authorization models for fine-grain access control. It defines a model-driven method to construct software which meets access control requirements and validates that the software does in fact have the necessary level of security. This was further studied in [DT-17, JP-11], where the inclusion of access control features in the UMM specification of components was investigated. Also, this incorporation of the access control in the multi-level interface was used in the discovery and selection of components. The access control properties of an ensemble of components, based on the individual access control features, were predicted. This was based on the principles of logic programming and the logic of temporal actions.

d. The initial exploration of the applicability of the UniFrame principles to the distributed real-time and embedded (DRE) systems was started during FY 2003. Since a UniFrame approach to constructing such systems would necessarily entail many different possible compositions, techniques based on genetic algorithms and Petri nets to prune this set of combinations to achieve a better QoS assurance were developed [CW-48, CW-49, CW-50].

e. Implementation of the first version of visual meta-programming language was completed [CW-2, CW-3, DT-14]. An original approach to the software monitoring automation based on precise behavior models and event grammars was developed. This allows an implementation of different kinds of monitoring, such as assertion checking, profiling, performance measurement, debugging queries, software visualization, intrusion detection, and dynamic QoS metrics within a uniform framework [CW-10, CW-20, CW-27, CW-34, CW-42, CW-51, CW-52, CW-53, DT-8, DT-18].

f. The design of the prototype for an automated test generator for reactive and real time systems based on attributed event grammars was completed. Efforts to assess the effectiveness of the tool were started in FY 2003. This approach provides for new tools for automated test driver generation and system safety assessment [CW-51, CW-52, CW-53, DT-22].

g. The design of a run-time monitoring tool for C/C++ programs based on the Dyninst instrumentation tool was also created. The exploration with this tool on different test scenarios was initiated in FY 2003.

h. Also, the feasibility of reactive system prototype verification based on Statechart models and build-in temporal logic assertion checking was explored [CW-52].

i. The design of the prototype for an automated test generator for reactive and real time systems based on attributed event grammars was completed. Efforts to assess the effectiveness of the tool were started in FY 2003.

Dissemination

The results of the UniFrame research were published and presented in various professional forums such as journals, conferences, workshops, showcases and seminars during the entire duration of the project. The details of these are provided below. In addition, a website (www.cs.iupui.edu/uniFrame) was created and maintained for the dissemination of the research results. The copies of a few sample publications are enclosed with this report.

Publications

Book Chapters [BC]

1. Andrew M. Olson, Rajeev R. Raje, Barrett R. Bryant, Mikhail Auguston, Carol Burt, "UniFrame -- A Unified Framework for Developing Service-oriented, Component-based Distributed Software Systems", in Service-Oriented Software System Engineering: Challenges and Practice (Eds: Stojanovic and Dahanayake), pp: 68-87, Idea Group Publishing, 2005.

2. Graham Wilcock, Paul Buitelaar, Antonio Pareja-Lora, Barrett Bryant, Jimmy Lin, Nancy Ide, "The Roles of Natural Language and XML in the Semantic Web", in Computational Linguistics and Beyond (Eds: Huang and Lenders), pp. 139-186, Institute of Linguistics, Academia Sinica, Tamkang, Taiwan, 2004.

3. Andrew M. Olson, Rajeev R. Raje, Barrett R. Bryant, Mikhail Auguston, Carol Burt, "A Process for Generating Software for Distributed, Heterogeneous Systems", (invited) in Parallel and Distributed Computing: Evaluation, Improvement and Applications (Eds: Y.S. Dai, Y. Pan, and R. Raje), Nova Science Publishers, 2005, (To Appear).

Journal Papers [JP]

1. Rajeev R. Raje, Barrett Bryant, Mikhail Auguston, Andrew Olson, Carol Burt, "A QoS-based Framework for Creating Distributed and Heterogeneous Software Components", Concurrency and Computation: Practice and Experience: 2002, 14, pp: 1009-1034, 2002.

2. Lee, Beum-Seuk and Bryant, Barrett R., "Applying XML Technology for Implementation of Natural Language Specifications," International Journal of Computer Systems, Science and Engineering 5 (September 2003), 3-24.

3. Chunmin Yang, Barrett R. Bryant, Carol Burt, Rajeev R. Raje, Andrew M. Olson, Mikhail Auguston, "Formal Methods for Quality of Service Analysis in Component-based Distributed Computing", Journal of Design & Process Science: Transactions of the Society for Design and Process Science, 8, 2, pp. 137-149, 2004.

4. Fei Cao, Barrett Bryant, Rajeev R. Raje, Andrew Olson, Mikhail Auguston, Wei Zhao, Carol Burt, "A Component Assembly Approach Based on Aspect-oriented Generative Domain Modeling", Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier Science, Vol. 114, pp. 119-136, 2005.

5. Fei Cao, Barrett Bryant, Rajeev R. Raje, Andrew Olson, Mikhail Auguston, Wei Zhao, Carol Burt, "A Non-Invasive Approach to Assertive and Autonomous Dynamic Component Composition in Service-Oriented Paradigm", Journal of Universal Computer Science, (Invited – Under Review) 2005.

6. Fei Cao, Barrett R. Bryant, Wei Zhao, Carol C. Burt, Rajeev R. Raje, Andrew M. Olson, Mikhail Auguston, "Model-Driven Reengineering Legacy Software Systems to Web Services", International Journal of Information Technology and Web Engineering (Under Review), 2005.

7. Pradeep Mysore, Rajeev R. Raje, Barrett R. Bryant, Purushotham Bangalore, "Building High-performance Systems using GridFrame", International Journal of High-performance Computer Applications (Invited – To be Submitted), 2005.

8. Wei Zhao, Barrett R. Bryant, Fei Cao, Rajeev R. Raje, Mikhail Auguston, Carol C. Burt, Andrew M. Olson, "The Language Oriented Domain Analysis Method," Computer Languages, Systems and Structures (To be Submitted), 2005.

9. Rajeev R. Raje, Anjali Kumari, Andrew Olson, Barrett Bryant, Mikhail Auguston, Carol Burt, "Multi-level Specification Matching", Concurrency and Computation (To be Submitted), 2005.

10. Rajeev R. Raje, Barun Devaraju, Pradeep Mysore, Andrew Olson, Barrett Bryant, Mikhail Auguston, Carol Burt, "Incorporating Selective Search and Customization in the UniFrame Resource Discovery Service", Cluster Computing (To be Submitted), 2005.

11. Alex Crespi, Rajeev R. Raje, Carol Burt, Andrew Olson, Barrett Bryant, Mikhail Augiston, "Access Control in UniFrame", IEEE Transactions on Parallel and Distributed Systems (To be Submitted), 2005.

Conference/Workshop Papers [CW]

1. Rajeev R. Raje, Barrett Bryant, Mikhail Auguston, Andrew Olson, Carol Burt, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components", Proceedings of the 2001 Monterey Workshop (Sponsored by DARPA, ONR, ARO and AFOSR), pp: 109-199, Monterey, California, 2001.

2. Mikhail Auguston, "Visual Meta-Programming Notation", Proceedings of the 2001 Monterey Workshop (Sponsored by DARPA, ONR, ARO and AFOSR), pp: 50-61, Monterey, California, 2001.

3. Mikhail Auguston, Valdis Berzins, Barrett Bryant, "Visual Meta-Programming Language", Proceedings of the OOPSLA 2001 Workshop on Domain Specific Visual Languages, pp: 69-82,Tampa Bay, Florida, 2001.

4. Girish Brahnmath, Rajeev R. Raje, Andrew Olson, Barrett Bryant, Mikhail Auguston, Carol Burt, "A Quality of Service Catalog for Software Components", Proceedings of the Southeastern Software Engineering Conference}, pp: 513-520,Huntsville, Alabama, 2002.

5. Barrett R. Bryant, Beum-Seuk Lee, "Two-Level Grammar as an Object-Oriented Requirements Specification Language", Proceedings of the 35th Hawaii International Conference on System Sciences, CD-ROM, 10 pages, Hawaii, 2002.

6. Beum-Seuk Lee, Barrett Bryant, "Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language", Proceedings of the 2002 ACM Symposium on Applied Computing, pp: 932-936, Madrid, Spain, 2002.

7. Carol Burt, Rajeev R. Raje, Mikhail Auguston, Barrett Bryant, Andrew Olson, "Quality of Service (QoS) Standards for Model Driven Architecture", Proceedings of the Southeastern Software Engineering Conference, pp: 521-529, Huntsville, Alabama, 2002.

8. Beum-Seuk Lee, Barrett Bryant, "Prototyping of Requirements Documents Written in Natural Language", Proceedings of SESEC 2002, the Southeastern Software Engineering Conference, pp: 538-543, Huntsville, Alabama, 2002.

9. Beum-Seuk Lee, Barrett Bryant, "Contextual Knowledge Representation for Requirements Documents in Natural Language", Proceedings of FLAIRS 2002, the 15th International Florida AI Research Symposium, pp: 370-374, Pensacola Beach, Florida, 2002.

10. J. Bret Michael, Mikhail Auguston, N. Rowe, R. Riehle, "Software Decoys: Intrusion Detection and Countermeasures", Proceedings of the IEEE Workshop on Information Assurance, United States Military Academy, pp: 130-138, West Point, New York, 2002.

11. Chunmin Yang, Barrett Bryant, Rajeev Raje, Mikhail Auguston, Andrew Olson, Carol Burt, "Formal Specification in Heterogeneous Distributed Software Integration", Proceedings of the 40th Annual ACM Southeast Conference, pp. 201-202, Raleigh, North Carolina, 2002.

12. Fei Cao, Barrett Bryant, Rajeev Raje, Mikhail Auguston, Andrew Olson, Carol Burt, "Specifying Heterogeneous Distributed Components", Proceedings of the 40th Annual ACM Southeast Conference, pp: 199-200, Raleigh, North Carolina, 2002.

13. Wei Zhao, Barrett Bryant, Rajeev Raje, Mikhail Auguston, Andrew Olson, Carol Burt, "A Unified Approach to Component Assembly Based on Generative Programming", Online Proceedings of the Workshop on Generative Programming, Austin, Texas, 2002.

14. Wei Zhao, Barrett Bryant, Rajeev Raje, Mikhail Auguston, Andrew Olson, Carol Burt, "Generative Composition of Distributed and Heterogeneous Components", Proceedings of the 40th Annual ACM Southeast Conference, pp: 195-196, Raleigh, North Carolina, 2002.

15. Nanditha Siram, Rajeev Raje, Barrett Bryant, Andrew Olson, Mikhail Auguston, Carol Burt, ``An Architecture for the UniFrame Resource Discovery Service", Proceedings of the 3rd International Workshop on Software Engineering and Middleware, pp: 20-35, Orlando, Florida, 2002.

16. Barrett Bryant, Mikhail Auguston, Rajeev Raje, Andrew Olson, Carol Burt, "Formal Specification of Generative Component Assembly Using Two-Level Grammar", Proceedings of the SEKE 2002, Fourteenth International Conference on Software Engineering and Knowledge Engineering, pp: 209-212, Ischia, Italy, 2002.

17. Wei Zhao, "A Product Line Architecture for Component Model Domains", Proceedings of PhDOOS 2002, Online Proceedings of the 12th Workshop for PhD Students in Object-Oriented Systems, Malaga, Spain, 2002.

18. Beum-Seuk Lee, Barrett Bryant, "Contextual Processing and DAML for Understanding Software Requirements Specifications", Proceedings of COLING 2002, the 19th International Conference on Computational Linguistics, pp: 516-522,Taipei, Taiwan, August 2002.

19. Carol C. Burt, Barrett R. Bryant, Rajeev R. Raje, Andrew Olson. Mikhail Auguston, "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models", Proceedings of the 6th IEEE International Enterprise Distributed Object Computing Conference, pp: 212-223, Lausanne, Switzerland, 2002.

20. Mikhail Auguston, C. Jeffery, S. Underwood, "A Framework for Automatic Debugging", Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE 2002, pp: 217-222, Edinburgh, U.K., 2002.

21. Fei Cao, "Using Two-Level Grammar in Component Specification", Proceedings of the First ACM SIGPLAN Conference on Generators and Components (GPCE 2002), Young Researchers Workshop, Pittsburgh, Pennsylvania, 2002.

22. Wei Zhao, "Two-Level Grammar as the Formalism for Middleware Generation in Internet Component Broker Organizations", Online Proceedings of the First ACM SIGPLAN Conference on Generators and Components (GPCE 2002), Young Researchers Workshop, Pittsburgh, Pennsylvania, 2002.

23. Changlin Sun, Rajeev Raje, Andrew Olson, Barrett Bryant, Mikhail Auguston, Carol Burt, Zhisheng Huang, "Composition and Decomposition of Quality of Service Parameters in Distributed Component-Based Systems", Proceedings of the IEEE 5th International Conference on Algorithms and Architectures for Parallel Processing, pp: 273-277, Beijing, China, 2002.

24. Zhisheng Huang, Rajeev Raje, Andrew Olson, Barrett Bryant, Mikhail Auguston, Carol Burt, Changlin Sun, "System-Level Generative Programming of Unified Approach Based on UMM for the Integration of Distributed Software Components", Proceedings of the IEEE 5th International Conference on Algorithms and Architectures for Parallel Processing, Beijing, China, 2002.

25. Fei Cao, Barrett Bryant, Rajeev Raje, Mikhail Auguston, Andrew Olson, Carol Burt, "Component Specification and Wrapper/Glue Code Generation with Two-Level Grammar using Domain Specific Knowledge", Proceedings of the Proceedings of ICFEM 2002, 4th International Conference on Formal Engineering Methods, pp: 136-142, Shanghai, China, 2002.

26. Beum-Seuk Lee, Barrett Bryant, "Automation of Software System Development Using Natural Language Processing and Two-Level Grammar", Proceedings of the Monterey Workshop, pp: 244-257, Venice, Italy, 2002.

27. Clinton Jeffery, Mikhail Auguston, S. Underwood, "Towards Fully Automatic Execution Monitoring", Proceedings of the Monterey Workshop, pp: 232-243, Venice, Italy, 2002.

28. Chunmin Yang, Beum-Seuk Lee, Barrett Bryant, Carol Burt, Rajeev Raje, Andrew Olson, "Formal Specification of Non-Functional Aspects in Two-Level Grammar", UML 2002 Workshop on Component-Based Software Engineering and Modeling Non-Functional Aspects, Dresden, Germany, 2002.

29. Wei Zhao, Barrett R. Bryant, Rajeev R. Raje, Mikhail Auguston, Andrew M. Olson, Carol C. Burt, "A Component Assembly Architecture with Two-Level Grammar Infrastructure", Online Proceedings of the OOPSLA'2002 Workshop on Generative Techniques in the context of MDA, Seattle, Washington, 2002.

30. Purvi Shah, Barrett R. Bryant, Rajeev R. Raje, Carol Burt, Andrew Olson, Mikhail Auguston, "Interoperability between Mobile Distributed Components using the UniFrame Approach", Proceedings of the 41st Annual ACM South East Conference, pp: 30-35, Savannah, GA, 2003.

31. Natasha Gupta, Rajeev R. Raje, Andrew Olson, Barrett Bryant, Mikhail Auguston, Carol Burt, "Analyzing the Web Services and UniFrame Paradigms", CD-ROM Proceedings of the Southeastern Software Engineering Conference (8 pages), Huntsville, Alabama, 2003.

32. Carol C. Burt, Rajeev R. Raje, Barrett R. Bryant, Andrew Olson, Mikhail Auguston, "Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control", Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference, pp: 159-173, Brisbane, Australia, 2003.

33. Wei Zhao, Barrett R. Bryant, Jeff Gray, Carol C. Burt, Rajeev R. Raje, Mikhail Auguston, Andrew M. Olson, "A Generative and Model Driven Framework for Automated Software Product Generation", Proceedings of the 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction, pp: 103-108, Portland, Oregon, 2003.

34. Clinton Jeffery, Mikhail Auguston, "Some axioms and issues in the UFO dynamic analysis framework", Proceedings of Workshop on Dynamic Analysis, ICSE'03, 25th International Conference on Software Engineering, pp: 45-48, Portland, Oregon, 2003.

35. Fei Cao, Barrett Bryant, Carol Burt, Jeffrey Gray, Rajeev Raje, Andrew Olson, Mikhail Auguston, "Modeling Web Services: Towards System Integration in UniFrame", Proceedings of the 7th World Conference on Integrated Design and Process technology (IDPT 2003), pp: 83-91, Austin, Texas, 2003.

36. Chunmin Yang, Barrett Bryant, Carol Burt, Rajeev R. Raje, Andrew Olson, Mikhail Auguston, "Formal Methods for Quality of Service Analysis in Component-Based Distributed Computing", Proceedings of the 7th World Conference on Integrated Design and Process technology (IDPT 2003), pp: 291-299, Austin, Texas, 2003.

37. Fei Cao, Barrett R. Bryant, Carol C. Burt, Zhisheng Huang, Rajeev R. Raje, Andrew M. Olson, Mikhail Auguston, "Automating Feature-Oriented Domain Analysis", Proceedings of the International Conference on Software Engineering Research and Practice (SERP'03), pp: 944-949, Las Vegas, Nevada, 2003.

38. Barrett Bryant, Beum-Seuk Lee, Fei Cao, Wei Zhao, Carol Burt, Rajeev Raje, Andrew Olson, Mikhail Auguston, "From Natural Language Requirements to Executable Models of Software Components", Proceedings of the 2003 Monterey Workshop, pp: 51-58, Chicago, Illinois, 2003.

39. Beum-Seuk Lee, Xiaoqing Wu, Fei Cao, Shih-hsi Liu, Wei Zhao, Chunmin Yang, Barrett R. Bryant, Jeffrey G. Gray, "T-Clipse: an Integrated Development Environment for Two-Level Grammar", OOPSLA 2003 Workshop on Eclipse Technology eXchange, pp: 91-95, Anaheim, California, 2003.

40. Fei Cao, Barrett Bryant, Rajeev Raje, Mikhail Auguston, Andrew Olson, Carol Burt, "Assembling Components with Aspect-oriented Modeling/Specification", Proceedings of UML Workshop W2 -- Workshop in Software Model Engineering WiSE@UML'2003), (Online), Francisco, California, 2003.

41. Fei Cao, Barrett Bryant, Rajeev R. Raje, Mikhail Auguston, Andrew Olson, Carol Burt. "A Component Assembly Approach Based on Aspect-Oriented Generative Domain Modeling", Proceedings of SC'04, Software Composition Workshop affiliated with ETAPS 2004, (Online), Barcelona, Spain, 2004.

42. Mikhail Auguston, Mark Trakhtenbrot, Run Time Monitoring of Reactive System Models, in Proceedings of Second International Workshop on Dynamic Analysis WODA 2004, the 26th International Conference on Software Engineering ICSE 2004, pp: 68-75, Edinburgh, Scotland, 2004,

43. Wei Zhao, Barrett R. Bryant, Fei Cao, Rajeev R. Raje, Mikhail Auguston, Carol C. Burt, Andrew M. Olson, "Grammatically Interpreting Feature Compositions", Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE'04), pp: 185-191. Banff, Canada, 2004.

44. Fei Cao, Barrett Bryant, Carol Burt, Rajeev R. Raje, Andrew Olson, Mikhail Auguston, "A Meta-modeling Approach to Web Services", Proceedings of ICWS04, IEEE International Conference on Web Services, pp: 796-799, San Diego, California, 2004.

45. Wei Zhao, Barrett R. Bryant, Rajeev R. Raje, Mikhail Auguston, Carol C. Burt, Andrew M. Olson, "Automated Glue/Wrapper Code Generation in Integration of Distributed and Heterogeneous Software Components", Proceedings of the 8th IEEE Enterprise Distributed Computing Systems Conference (EDOC'04), pp: 275-285, Monterey, California, 2004.

46. Pradeep Mysore, Rajeev R. Raje, Purushottam Banglore, Barrett Bryant, "GridFrame -- A Framework for Building Component-based Grid Systems", Proceedings of 12th International Conference on Advanced Computing & Communication (ADCOM '04), pp: 23-31, Ahmedabad, India, 2004.

47. Fei Cao, Barrett Bryant, Wei Zhao, Carol Burt, Rajeev R. Raje, Andrew Olson, Mikhail Auguston. "Marshaling and Unmarshaling Models Using Entity-Relationship Model", Proceedings of ACM SAC'05, ACM Symposium on Applied Computing, pp: 1553-1557, Santa Fe, New Mexico, 2005.

48. Shih-hsi Liu, Barrett Bryant, Jeffrey Gray, Rajeev R. Raje, Andrew Olson, Mikhail Auguston. "Two-level Assurance of QoS Requirements for Distributed Real-time and Embedded Systems", Proceedings of ACM Symposium on Applied Computing, SAC'05, pp: 903-904, Santa Fe, New Mexico, 2005.

49. Shih-hsi Liu, Barrett R. Bryant, Jeffrey G. Gray, Rajeev Raje, Andrew Olson and Mikhail Auguston, "QoS-UniFrame: A Petri Net-based Modeling Approach to Assure QoS Requirements of Distributed Real-time and Embedded Systems", Proceedings of the 12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'05), pp: 202-209, Greenbelt, Maryland, 2005.

50. Shih-hsi Liu, Fei Cao, Barrett R. Bryant, Jeffrey G. Gray, Rajeev Raje, Andrew Olson and Mikhail Auguston, "Quality of Service-Driven Requirements Analyses for Component Composition: A Two-Level Grammar Approach", To appear in the Proceedings of the17th

International Conference on Software Engineering and Knowledge Engineering (SEKE'05), Taipei, Taiwan, 2005.

51. Mikhail Auguston, James Bret Michael, Man-Tak Shing, Environment Behavior Models for Scenario Generation and Testing Automation, in Online Proceedings of the First International Workshop on Advances in Model-Based Software Testing (A-MOST'05), The 27th International Conference on Software Engineering ICSE'05, 2005, St. Louis, 2005.

52. Mikhail Auguston, James Bret Michael, Man-Tak Shing, Test Automation and Safety Assessment in Rapid Systems Prototyping, to appear in Proceedings of 16th IEEE International Workshop on Rapid System Prototyping, pp: 188-194, Montreal, Canada, 2005.

53. Mikhail Auguston, James Bret Michael, Man-Tak Shing, and David L. Floodeen, "Using Attributed Event Grammar Environment Models for Automated Test Generation and Software Risk Assessment of System-of-Systems", To appear in the Proceedings of 2005 IEEE International Conference on Systems, Man, and Cybernetics, Special Session on Recent Advances in Engineering Systems-of-Systems to Support Joint and Coalition Warfighters, The Big Island, Hawaii, 2005.

Standards Documents [SD]

1. Carol C. Burt, "mars/04-04-15: Business Model Driven Access Management for Service-Oriented Applications", OMG Draft RFP.

2. Carol C. Burt, "mars/04-02-12: Model Driven Access Management", OMG Draft RFP for a Platform Independent Model for Access Management (with mappings to existing PSMs such as OASIS XACML, OMG RAD, JCP JAAS, Microsoft Authorization Manger.

Dissertations and Theses [DT]

1. Nanditha N. Siram, "An Architecture for Discovery of Heterogeneous Software Components.", M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, May, 2002.

2. Christina Varghese, "Examining, Documenting, and Modeling the Problem Space of a Variable Domain", M. S. Project, TR-CIS-0612-02, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, August, 2002.

3. Girish Brahnmath, "The UniFrame Quality Of Service Framework", M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, December, 2002.

4. Zhisheng Huang, "The UniFrame System-Level Generative Programming Framework", M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, May, 2003.

5. Changlin Sun, "QoS Composition and Decomposition in UniFrame", M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, August, 2003.

6. Robert Berbeco, "The UniFrame .NET Web Service Discovery Service", M. S. Project, TR-CIS-0630-03, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, August, 2003.

7. Beum Seuk Lee, "Automated Conversion from a requirements document to an executable formal specification using two-level grammar and contextual natural language processing", Ph. D. Dissertation, Department of Computer and Information Sciences, The University of Alabama at Birmingham, August 2003.

8. Tee Huu Saw, Captain (Singapore Armed Forces), "Evaluation of a Multi-agent System for Simulation and Analysis of Distributed Denial-of-Service Attacks", Department of Computer Science, Naval Postgraduate School, June 2003.

9. Richard M. Neidermyer, "Unified Meta-Component Model Specification Editor", M.S. Project TR-CIS-0330-04, Department of Computer & Information Science, Indiana University Purdue University, May, 2004.

10. Kalpana Tummala, "Glue Generation Framework In UniFrame for the CORBA-JAVA/RMI Interoperability", M.S. Project TR-CIS-0302-03, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, May, 2004.

11. Jayasree Gandhamaneni, "UniFrame Mobile Agent Based Resource Discovery Service (MURDS)", M.S. Project TR-CIS-1122-03, Department of Computer & Information Science, Indiana University Purdue University, August 2004.

12. Praveen Gopalakrishna, "Modeling QOS Parameters In Component-Based Systems", M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, August, 2004.

13. Natasha S. Gupta, "An Exploratory Analysis of the .NET Component Model and UniFrame paradigm using a collaborative approach", M. S. Thesis , Department of Electrical and Computer Engineering, Indiana University Purdue University Indianapolis, August, 2004.

14. Graham C. Pierson, Major (US Marine Corps), "Code Maintenance and Design for a Visual Programming Language Graphical User Interface", M. S. Thesis, Department of Computer Science, Naval Postgraduate School, September 2004.

15. Anjali Kumari, "Synchronization and Quality of Service Specification and Matching of Software Components", M. S. Thesis, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, December, 2004.

16. Padmavathi Kambhampati, "UML Variability and Automation of Variant Models", M. S. Thesis, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, December, 2004.

17. Alexander Crespi, "An Access Control Model for the UniFrame Framework", M. S. Thesis, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, December, 2004.

18. Yuan Chen, Major (Singapore Navy), "Evaluation of a Multi-agent System for Simulation and Analysis of Distributed Denial-of-Service Attacks", M. S. Thesis, Department of Computer Science, Naval Postgraduate School, December 2004.

19. Barun Devaraju, "Enhancement of The UniFrame Resource Discovery Service", M. S. Thesis, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, May, 2005.

20. Srikanth Reddy, "UniFrame Resource Discovery Service Monitoring and Management System", M. S. Project, TR-CIS-0411-05, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, May, 2005.

21. Pradeep Mysore, "An Experimental Evaluation of UniFrame Resource Discovery System", M. S. Thesis, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, May, 2005.

22. James Imanian, LCDR (US Navy), "Automatic Test Case Generation for Reactive Software Systems Based on Environment Models", M. S. Thesis, Department of Computer Science, Naval Postgraduate School, June 2005.

23. Fei Cao, "Model-Driven Development and Dynamic Composition of Web Services", Ph. D. Dissertation, Department of Computer and Information Sciences, The University of Alabama at Birmingham, July 2005.

24. Wei Zhao, Ph. D. Dissertation, "Transformations from Business Process Models to High Level Programming Languages", Department of Computer and Information Sciences, The University of Alabama at Birmingham, December 2005.

Invited Presentations

(Apart from the conference and workshop presentations)

1. Rajeev R. Raje, "UniFrame", Software Engineering Research Center Summer Showcase, Indianapolis, Indiana, June 2001.

2. Rajeev R. Raje, "UniFrame", CIP/SW Kickoff Meeting. Washington, D.C., July 2001.

3. Carol C. Burt, Barrett R. Bryant, Rajeev R. Raje, "UniFrame", OMG Meeting. Toronto, Ontario, Canada, September 2001.

4. Barrett R. Bryant, "Object-Oriented Natural Language Requirements Specification", Sun Yat-Sen University, Guangzhou, China, United Nations University, International Institute for Software Technology, Macau, China, and Chinese University of Hong Kong, Shatin, Hong Kong, China, October 2001.

5. Andrew Olson, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", SERC Showcase, Muncie, Indiana, December 2001.

6. Rajeev R. Raje, "UniFrame", at the Connect-Tech, Indianapolis, IN, December 2001.

7. Carol Burt, "UniFrame - a unified framework for integration of distributed components", OMG Technical Meeting (to ORB/OS Group), Anaheim, California, February 2002.

8. Barrett R. Bryant, "XML and DAML for Contextual Knowledge Representation of Natural Language Requirements Documents", 79th Annual Meeting of the Alabama Academy of Science, Livingston, Alabama, March 2002.

9. Fei Cao, "Locating Heterogeneous Distributed Components Using Headhunters", 79th Annual Meeting of the Alabama Academy of Science, Livingston, Alabama, March 2002. (This presentation was co-winner of the Student Research Award for best student presentation in the Engineering and Computer Science Section of the Alabama Academy of Science.)

10. Chunmin Yang, "Application of Formal Methods in Distributed Computing", 79th Annual Meeting of the Alabama Academy of Science, Livingston, Alabama, March 2002. (This presentation was co-winner of the Student Research Award for best student presentation in the Engineering and Computer Science Section of the Alabama Academy of Science.)

11. Wei Zhao, "Generative Automation of Middleware", 79th Annual Meeting of the Alabama Academy of Science, Livingston, Alabama, March 2002.

12. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components," University of Houston, Houston, Texas, April, 2002.

13. Barrett R. Bryant, "Object-Oriented Natural Language Requirements Specification", University of Milan at Crema, Crema, Italy, and Soft People Tecnologie.net, Milan, Italy, April 2002.

14. Natasha Gupta, "Encompassing .Net Framework and Web Services into UniFrame", Spring Showcase of SERC, Morgantown, VW, May 2002.

15. Carol C. Burt, "UniFrame", University of Edinburgh, Edinburgh, Scotland, United Kingdom, May 2002.

16. Rajeev R. Raje, "UniFrame," U. S. Office of Naval Research, London, England, United Kingdom, May, 2002.

17. Barrettt R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components," University of Lancaster, Lancaster, England, United Kingdom, May 2002.

18. Barrett R. Bryant, "Object-Oriented Natural Language Requirements Specification",Carlos III University of Madrid, Madrid, Spain, June 2002.

19. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", INRIA, Sophia Antipolis, France, June 2002.

20. Barrett R. Bryant, "Object-Oriented Natural Language Requirements Specification", University of Ljubljana, Ljubljana, Slovenia, July 2002.

21. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", National Taiwan University, Taipei, Taiwan, August 2002.

22. Rajeev R. Raje, "UniFrame", NSF ES-EU Workshop, Landsdowne, VA, September 2002.

23. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", Illinois Institute of Technology, Chicago, Illinois, October 2002.

24. Rajeev R. Raje, "UniFrame", VJTI, University of Bombay, India, December 2002.

25. Natasha Gupta, Girish Brahnmath, "UniFrame and Component Quality of Service", SERC Fall Showcase, Muncie, Indiana, December 2002.

26. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", Magic City Java Users Group, Birmingham, Alabama, February, 2003.

27. Andrew Olson, "Herding Software Development Over the Electronic Range", Department of Computer & Information Science, IUPUI, March, 2003.

28. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", Jagiellonian University, Krakow, Poland, April, 2003.

29. Rajeev R. Raje, Barrett Bryant, Mikhail Auguston, "UniFrame", U. S. Office of Naval Research (Annual Review), Harpers Ferry, West Virginia, May 2003.

30. Alex Crespi, Praveen Gopalakrishna, "UniFrame Discovery Service and the System Generator", Spring 2003 Software Engineering Research Center Showcase, West Virginia University, Morgantown, West Virginia, May 2003.

31. Rajeev R. Raje, Barrett R. Bryant, Andew M. Olson, Mikhail Auguston, Carol C. Burt, "UniFrame", U. S. Office of Naval Research (Final Review), Annapolis Junction, Maryland, November 2003.

32. Carol Burt, "Model Driven Access Management", OMG's Security Information Day, London, U. K., November 2003.

33. Andrew M. Olson, "Constructing Distributed Computing Systems with the UniFrame Process", Departamento de las Ciencias de la Computación, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Santiago, Chile, December 2003.

34. Carol Burt, "Model Driven Access Management", OMG's Security Information Day, Anaheim, California, January 2004.

35. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", ACM Student Chapter, University of Porto, Porto, Portugal, March 2004.

36. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", University of Minho, Braga, Portugal, March 2004.

37. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", New University of Lisbon, Lisbon, Portugal, March 2004.

38. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", NASA Ames Research Laboratory, Mountain View, California, September 2004.

39. Amruta Jejurikar, Rajeev R. Raje, "Development of Distributed Component-based Systems – A Symbiosis of UniFrame Principles and Microsoft's Infrastructure", Microsoft Corporation, Redmond, Washington, June 2005.

40. Barrett R. Bryant, "UniFrame: Framework for Seamless Interoperation of Heterogeneous Distributed Software Components", University of Maribor, Slovenia, and University of Milan at Crema, Italy, June 2005.

41. Rajeev R. Raje, "UniFrame Resource Discovery Service", The University of Alabama at Birmingham and IEEE Section of Birmingham, July 2005 (Scheduled).

Prototypes

1. URDS (UniFrame Resource Discovery Service) – Different Versions
2. URDS Management and Monitoring System
3. UniFrame System Constructor
4. UniFrame QoS Catalog
5. Glue and Wrapper Generators
6. Automated test generator for reactive and real time systems based on environment models
7. Run time monitoring system for C/C++ executables via Dyninst

Collaborations and Interactions

The UniFrame research team interacted with a variety of groups, academic and industrial organizations, during the investigations. The interactions were in the forms of discussions, seminars, joint publications and proposals. Below these partners are enumerated.

1. Academic Institutes: Michigan State University, Charles University (Czech Republic), University of Maribor (Slovenia), Lancaster University (UK), University of Edinburgh (UK), Jadavpur University (India), IIT-B (India), VJTI (India).

2. Industrial Organizations: 2AB, Inc., Microsoft Corporation, Stryon Incorporated, Disha Technologies, Inc., OMG (Object Management Group), SERC (Software Engineering Research Center), BBN Technologies, Computer Sciences Corporation.

Citations

The UniFrame research publications have been well received by the community. Many of these publications have been cited by other researchers. A citation search on Google indicated more than fifty external citations to various UniFrame related publications.

Educational Activities

Students

1. **IUPUI** – R. Berbeco, G. Brahnmath, R. Bulusu, A. Crespi, V. Cheekati*, B. Devaraju, J. Freeman, J. Gandhamaneni*, P. Gopalkrishna, N. Gupta*, J. Hansome, Z. Huang, A. Jejurikar*, A. Kumari*, P. Kambhampati*, P. Mysore, N. Nayani*, R. Neidermyer, S. Reddy, M. Ridzal, C. Sun, O. Tilak, K. Tummala*, C. Varghese*.

2. **UAB** – F. Cao, B. Lee, S. Liu, S. Mugala, R. Puljala, P. Shah*, X. Wu. C. Yang*, W. Zhao*.

3. **NMSU/NPS** – G. Fragkos, A. Islam, S. Underwood, T. Saw, G. Pierson, Y. Chen, J. Imanian.

*(Note: A majority of these students were financially supported by this award. The remaining students, although not financially supported, worked on research topics that stemmed from the UniFrame research. The women students are indicated with an * symbol. Students, at NPS, were employees of the DoD – US as well as allied countries.)*

Impact on Education

This research has impacted the education at all the participating institutes. The impact is classified into the four categories indicated below:

1. Enriching Student Research Experience by:
   a. Inter- and intra-university collaborations
   b. Proficiency with prevalent state-of-the-art
   c. Participation in professional forums

2. Impact on Curricula by:
   a. Incorporation of research material into courses at IUPUI, UAB, NMSU and NPS

3. Computing Infrastructure Enhancement by:
   a. Creation of Heterogeneous Computing Laboratories at IUPUI, UAB and NMSU

4. Invited Presentations to:
   a. Academic institutions, industrial forums and standards organizations

Other References [OR]

1. Raje, R., "UMM: Unified Meta-object Model for Open Distributed Systems", Proceedings of 4th IEEE International Conference on Algorithms and Architecture for Parallel Processing, ICA3PP'2000, pp: 454-465, 2000.

2. Beugnard, A., Jezequel, J., Plouzeau, N., Watkins, D., "Making Components Contract Aware", IEEE Computer, Vol. 32, No. 7, pp: 38-45, 1999.

3. Auguston, M., "Program Behavior Model Based on Event Grammar and its Application for Debugging Automaton", Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95), pp: 277-291, 1995.

4. Mukhopadhyay, S., Peng, S., Raje, R., Palakal, M., Mostafa J., "Multi-Agent Information Classification Using Dynamic Acquaintance Lists", Journal of the American Society for Information Science and Technology, Vol. 54(10), pp: 966-975, 2003. @article{Tha85,

5. Thathachar, M., Sastry, P., "A New Approach to the Design of Reinforcement Schemes for Learning Automata", IEEE Transactions on System Man Cybernetics, vol. 15, pp: 168-175, 1985.

6. Zaremski, A., Wing, J., "Specification Matching of Software Components, Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp: 6-17, 1995.

7. Zaremski, A., Wing, J., Specification Matching of Software Components, ACM Transactions on Software Engineering, vol. 6, no. 4, pp: 333-369, 1995.

8. Lamport, L., "Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison Wesley Publication Company, 2002.

9. Gamma, E., Helm R., Johnson, R., Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley Publication Company, 1994.

Appendix (List of Sample Publications)

Following papers are included as sample publications. The details (such as the venue of publication, dates, and authors are indicated earlier under the publications sections). The rest of the publications are available at the UniFrame website (www.cs.iupui.edu/uniFrame).

1. A Unified Approach for the Integration of Distributed Heterogeneous Software Components
2. Two-Level Grammar as an Object-Oriented Requirements Specification Language
3. A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components
4. A Quality of Service Catalog for Software Components
5. Quality of Service (QoS) Standards for Model Driven Architecture
6. An Architecture for the UniFrame Resource Discovery Service
7. Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models
8. A Framework for Automatic Debugging
9. Unified Approach for System-Level Generative Programming
10. Composition and Decomposition of Quality of Service Parameters in Distributed Component-based Systems
11. Automation of Software System Development Using Natural Language Processing and Two-Level Grammar
12. Formal Specification of Non-Functional Aspects in Two-Level Grammar
13. Towards Fully Automatic Execution Monitoring
14. A Component Assembly Architecture with Two-Level Grammar Infrastructure
15. Some Axioms and Issues in the UFO Dynamic Analysis Framework
16. Automating Feature-Oriented Domain Analysis
17. Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control
18. From Natural Language Requirements to Executable Models of Software Components
19. Assembling Components with Aspect-Oriented Modeling/Specification

20. Modeling Web Services: Towards System Integration in UniFrame

21. Automated Glue/Wrapper Code Generation in Integration of Distributed and Heterogeneous Software Components

22. Formal Specification of Generative Component Assembly Using Two-Level Grammar

23. Analyzing the Web Services and UniFrame Paradigms

24. A Component Assembly Approach Based on Aspect-Oriented Generative Modeling

25. QoS-UniFrame: A Petri Net-based Modeling Approach to Assure QoS Requirements of Distributed Real-time and Embedded Systems

26. Quality of Service-Driven Requirements Analyses for Component Composition: A Two-Level Grammar++ Approach

27. Marshaling and Unmarshaling Models Using the Entity-Relationship Model

28. A Meta-Modeling Approach to Web Services

29. Model-Driven Reengineering Legacy Software Systems to Web Services

30. A Non-Invasive Approach to Assertive and Autonomous Dynamic Component Composition in Service-Oriented Paradigm

31. GridFrame -- A Framework for Building Component-based Grid Systems

32. UniFrame -- A Unified Framework for Developing Service-oriented, Component-based Distributed Software Systems

(Appeared in the Proceedings of the *2001 Monterey Workshop* – Pages 109-119)

A Unified Approach for the Integration of Distributed Heterogeneous Software Components[1]

Rajeev R. Raje[2 3] Mikhail Auguston[4 5] Barrett R. Bryant[4 6] Andrew M. Olson[2] Carol Burt[7]

Abstract

Distributed systems are omnipresent these days. Creating efficient and robust software for such systems is a highly complex task. One possible approach to developing distributed software is based on the integration of heterogeneous software components that are scattered across many machines. In this paper, a comprehensive framework that will allow a seamless integration of distributed heterogeneous software components is proposed. This framework involves: a) a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glues and wrappers, based on a designer's specifications, for achieving interoperability, c) a formal mechanism for precisely describing the meta-model, and d) a formalization of quality of service (QoS) offered by each component and an ensemble of components. A case study from the domain of distributed information filtering is described in the context of this framework.

Keywords: Distributed systems, Formal methods, Glue and Wrapper technology, Quality of Service

1 Introduction

The rapid advances in the processor and networking technologies have changed the computing paradigm from a centralized to a distributed one. This change in paradigm is allowing us to develop distributed computing systems (DCS). DCS appear in many critical domains and are, typically, characterized by: a) a large number of geographically dispersed and interconnected machines, each containing a subset of the required data, b) an open architecture, c) a local autonomy over the hardware and software resources, d) a dynamic system configuration and integration, e) a time-sensitivity of the expected solution, and f) the quality of service with an appropriate notion of compensation. These characteristics make the software design of DCS an extremely difficult task.

One promising approach to the software design of DCS is based on the principles of distributed component computing. Under this paradigm DCS are created by integrating geographically scattered heterogeneous software components. These components constantly discover one another, offer/utilize services, and negotiate the cost and the quality of the services. Such a view provides a scalable solution and hides the underlying heterogeneity.

Various distributed component models, each with strengths and weaknesses, are prevalent and widely used. However, almost a majority of these models have been designed for 'closed' systems, i.e., systems, although distributed in nature, are developed and deployed in a confined setup. In contrast, a direct consequence of the heterogeneity, local autonomy and the open architecture is that the software realization of DCS requires combining components that adhere to different distributed models. This in turn increases the complexity of the design process of DCS. Hence, a comprehensive framework, that provides a seamless access to underlying components and aids in the design of DCS, is needed.

In this paper, one such framework is described. This framework consists of: a) a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glue and wrappers, based on a designer's specifications, for achieving interoperability, c) a formal mechanism for precisely describing the meta-model, and d) a formalization of the notion of quality of service offered by each component and an ensemble of components. The paper also presents a case study that shows the application of the framework to a specific problem domain.

The rest of the paper is organized as follows. The next section contains a detailed discussion about the meta-model. As an application of the meta model, a case study from the domain of distributed information filtering is presented in the Section 3. Section 4 deals with the formal specification of the meta model, the automated system integration, and evaluation of the approach. Finally, we conclude in Section 5.

[1]This material is based upon work supported by, or in part by, the U. S. Office of Naval Research under award number N00014-01-1-0746.

[2]Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 723 W. Michigan Street, SL 280, Indianapolis, IN 46202, USA, {rraje, aolson}@cs.iupui.edu, +1 317 274 5174/9733

[3]This material is based upon work supported by, or in part by, the National Science Foundation Digital Libraries Phase II grant.

[4]Computer Science Department, Naval Postgraduate School, 833 Dyer Rd., SP 517, Monterey, CA 93943, USA, {auguston, bryant}@cs.nps.navy.mil, +1 831 656 2509/2726

[5]This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number 40473-MA. On leave from Computer Science Department, New Mexico State University, USA.

[6]This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office r contract/grant number DAAD19-00-1-0350. On leave from Department of Computer and Information Sciences, University at Alabama at mingham, USA.

[7]2AB, Inc., 1700 Highway 31, Calera, AL 35040, USA, cburt@2ab.com, +1 205 621 7455

2 Component Models and a Meta-model

Many models and projects for the software realization of DCS have been proposed by academia and industry. A few prominent ones are: Java^{TM} Remote Method Invocation (RMI) [16], Common Object Request Broker Architecture (CORBA^{TM}) [16, 20], Distributed Component Object Model (DCOM^{TM}) [11, 16], Web-component model/DOM [10], Pragmatic component web [5], Hadas [6], Infospheres [4], Legion [22], and Globus [21]. Each of these models/projects has strength and weaknesses. Some of these are language-centric and only assume a uniform way of the world (Java); while the others allow a limited interoperability (CORBA - allowing implementations in different languages). Some of these are general-purpose, i.e., not concentrating on any particular application domain (DCOM), while others are specifically tailored to high-performance computing applications (Legion). However, almost all of these models/projects do not assume the presence of other models. Thus, the interoperability which they provide is limited mainly to the underlying hardware platform, operating system and/or implementational languages. Also, there are hardly any models which emphasize the notion of quality of service offered by the components. Projects, such as Agent TCL [8], etc., based on the principles of intelligent agents have imbibed the notion of the quality of service and related compensation. However, the agents are at a higher level of abstraction than components and many of the agent projects/frameworks use one or the other existing distributed-component models at the low-level.

2.1 Why a Meta-model?

Given the above mentioned plethora of component-based models and also noting the fact that components, by their definition, are independent of the implementation language, tools and the execution environment; it is necessary to answer the questions: *why is a meta-model needed for a seamless interoperation of distributed heterogeneous components?* and *how would a meta-model assist in seamlessly integrating distributed heterogeneous software components?* The answer to these question lies in: a) in any organization, software systems undergo changes and evolutions, b) local autonomy is an inherent characteristic of today's geographically (or logically) dispersed organizations, and c) if reliable software needs to be created for a DCS by combining components then the quality of service offered by each component needs to become a central theme of the software development approach.

The consequence of constant evolutions and changes is that there is a need to rapidly create prototypes and experiment with them in an iterative manner. Thus, there is no alternative but to adhere to cyclic (manual or semi-automatic) component-based software development for DCS. However, the solution of decreeing a common COTS environment, in an organization, is against the principle of local autonomy. Hence, the development of a DCS in an organization will, most certainly, require creating an ensemble of heterogeneous components, each adhering to some model. Also, every DCS is designed and developed with a certain goal in mind, and usually that goal is associated with a certain perception of the quality (as expected from the system) and related constraints.

Thus, there is a need for a comprehensive meta-model that will seamlessly encompass existing (and future) heterogeneous components by capturing their necessary aspects, including the quality of service offered by each component and an amalgamation of components.

2.2 Unified Meta-component Model (UMM)

In [17] we have proposed a unified meta-component model (UMM) for global-scale systems. The core parts of the UMM are: *components*, *service and service guarantees*, and *infrastructure*. The innovative aspects of the UMM are in the structure of these parts and their inter-relations. UMM provides an opportunity to bridge gaps that currently exist in the standards arena. For example, the CORBA Component Model (CCM^{TM}) [13] and Java Enterprise Edition component models (J2EE^{TM}) are consistent, and yet, because of the absence of a formal meta-model, it is difficult during the evolution of each to recognize when the boundaries that maintain the consistency are crossed. Similarly, it has been demonstrated in numerous products that the Component Object Model (COM^{TM}) [18] and CORBA component models are similar (in an abstract sense) enough to allow meaningful bridging. It is, however, not possible to point to a Meta-model that constrains the implementations of these technologies.

For enterprise component solutions, this is an area where significant standards work is now focused. The OMG Meta Object Facility (MOF^{TM}) [14] provides a common meta-model that allows the interchange of models between tools as well as the expression of models in XMI^{TM} (an MOF compliant XML^{TM} (eXtended Markup Language)) [12]. This work allows the generation of interfaces from Unified Modeling Language (UML) [19] models, however, a careful analysis of the resulting interface specifications makes it clear that distribution is not a key factor in the algorithms used. For example, quality of service requirements for performance, scalability and/or security would dictate the use of iterators, the factoring of interfaces to separate "query" and "administrative" operations, and the use of structures and/or objects passed by value. The current standards in this tend to focus on data access with accessors and mutators and relationship transversal. This is acceptable in a single machine environment, but unacceptable for highly distributed communications and collaborations. The recent shift in focus for the Object Management Group to "Model Driven Architecture" (MDA^{TM}) [15] is a recognition that to create mechanized software for the collaboration and bridging of component architectures will require standardization of Business and Component Meta-Models. The need to support the evolution of component models and to describe the capabilities of the models will be key to realizing the full potential of an E-business economy.

The following sections describe the various aspects of UMM in detail.

2.2.1 Component

In UMM, components are autonomous entities, whose implementations are non-uniform, i.e., each component adheres to some distributed-component model and there is no notion of either a centralized controller or a unified implementational framework. Each component has a state, an identity and a behavior. Thus, all components have well-defined interfaces and private implementations. In addition, each component in UMM has three aspects: 1) a computational aspect, 2) a cooperative aspect, and 3) an auxiliary aspect.

Computational Aspect

The computational aspect reflects the task(s) carried out by each component. It in turn depends upon: a) the objective(s) of the task, b) the techniques used to achieve these objectives, and c) the precise specification of the functionality offered by the component. In DCS, components must be able to 'understand' the functionality of other components. Thus, each component in UMM supports the concept of introspection, by which it will precisely describe its service to other inquiring components. There are various alternatives for a component to indicate its computation – ranging from simple text to formal descriptions. Both these extremes have advantages and drawbacks. UMM takes a mixed approach to indicate the computational aspect of a component – a simple textual part, called *inherent attributes* and a formal precise part, called *functional attributes.*

The functional part is formal and indicates precisely the computation, its associated contracts and the level(s) of service offered by the component. Multi-level contracts for components have been proposed by [2], classifying the contracts into four levels – syntactic, behavioral, concurrency and quality of service (QoS). UMM integrates this multi-level contract concept into the functional part of the computational aspect. As stated earlier, in DCS each component will be offering a service and hence, the level related to the QoS is especially critical in UMM. The QoS depends upon many factors such as, the algorithm used, the execution model, resources required, time, precision and classes of the results obtained. UMM makes an attempt at quantifying the QoS by creating a vocabulary and providing multiple levels of quality, which could be negotiated by the components involved in an interaction. The functional part will also be specified by the creator of the component.

Cooperative Aspect

In UMM, components are always in the process of cooperating with each other. This cooperation may be task-based or greed-based. The cooperative aspect depends on many factors: detection of other components, cost of service, inter-component negotiations, aggregations, duration, mode, and quality. Informally, the cooperative aspect of a component may contain: 1) Expected collaborators – other components that can potentially cooperate with this component, 2) Pre-processing collaborators – other components on which this component depends upon, and 3) Post-processing collaborators – other components that may depend on this component.

Auxiliary Aspect

In addition to computation and cooperation, mobility, security, and fault tolerance are necessary features of DCS. The auxiliary aspect of a component will address these features. In UMM, each component can be potentially mobile. The mobility of the component will be shown as a 'mobility attribute' (a notion similar to the inherent attribute). If a component is mobile, then the mobility attribute will contain the necessary information, such as its implementation details and required execution environment. Similarly, security in DCS is a critical issue. The security attribute of a component will contain the necessary information about its security features. As DCS are prone to frequent failures, full and partial, fault tolerance is critical in these systems. Similar to mobility and security, each component contains fault-tolerant attributes in its auxiliary aspect.

2.2.2 Service and Service Guarantees

The concept of a service is the second part of the UMM. A service could be an intensive computational effort or an access to underlying resources. In DCS, it is natural to have several choices for obtaining a specific service. Thus, each component, in addition to indicating its functionality, must be able to specify the cost and quality of the service offered.

The nature of the service offered by each component is dependent upon the computation performed by that component. In addition to the algorithm used, expected computational effort and resources required, the cost of each service will be decided by the motivation of the owner and the dynamics of supply and demand. In a dynamic environment costs must always be accompanied by the duration for which the costs are valid. As the system dynamics undergo constant changes, the methodologies used to fix the cost of a service will evolve as time progresses, thereby creating a need to indicate the time sensitiveness of the cost. The quality of service is an indication given by an component, on behalf of its owner, about its confidence to carry out the required services in spite of the constantly changing execution environment and a possibility of partial failures. The techniques used to determine the cost, the time-validity and the quality of a service will depend upon the tasks carried out by the component and the objectives of its owner and will involve principles of distributed decision making.

There are many parameters that a component can use to indicate its quality of service. A few examples are: i) Throughput - number of methods executed per second and classification of methods based on their read/write behaviors, ii) Parallelism constraints - synchronous or asynchronous, iii) Priority, iv) Latency or End-to-End Delay - turn-around time for an invocation, v) Capacity - how many concurrent requests a given component can handle, vi) Availability - indication of the reliability of a component, vii) Ordering constraints - can invocations (asynchronous) be executed out of order by a component, viii) Quality of the result returned - does the component provide a classification or ranking of the result, and ix) Resources available - how many resources (hardware/data) are accessible to the component under consideration and what are the types of resources.

When a component uses certain metrics to indicate its QoS (either all the mentioned criteria or a sub/super set of them), three interesting issues need to be addressed: a) how does the component developer decide these parameters?, b) how does the developer guarantee the advertised QoS during the execution?, and c) when components are collected together as a solution for specific DCS, what happens to the QoS of the combination and how does the combined QoS meet the quality requirements of DCS?

The parameters to be used to describe the QoS of a component are highly context (application) dependent. The proposed approach is to create lists of QoS metrics for common application domains. A few examples of such domains are: scientific computing, multi-media applications, information filtering, and databases. Once such lists are created, they would be used as a template by the component developers while advertising the QoS of their components.

QoS of Components

The issue of guaranteeing a particular QoS, for a component, in an ever changing dynamic DCS is extremely critical; mainly because of external (e.g., policy matters related to resources) and internal (e.g., changes in algorithms) factors that affect a life cycle of a component. In addition, as the software realization of DCS is based on an amalgamation of heterogeneous components, a proper guarantee of a QoS offered by a component effectively decides the QoS of the entire DCS. The quality metrics are expected to vary from one application domain to another and which metrics to select would depend on the intentions of the component developer and the functionality offered by that component. A few examples of such QoS metrics are already mentioned in the previous section. Irrespective of the metrics selected, there is a need for a well-defined mechanism that will assist the developer to achieve the necessary QoS when that component is deployed. Just like any software development process, the process of guaranteeing a certain QoS, as offered by a component, will be an incremental and iterative one, as will be discussed later.

QoS of an Integrated System

In addition to the QoS of individual components, there is a need to achieve a certain QoS for the ensemble of heterogeneous components assembled for a distributed system under discussion. The QoS of such an amalgamation will be decided by the design constraints of the system under construction. However, the integral characteristics of such a system typically cannot be expressed as a function of individual components but as a property of the whole system behavior. Hence, there is a need for a formal model of system behavior, which will integrate the behaviors of each component in the ensemble along with its QoS guarantees.

The proposed approach to address the problem of QoS is as follows. First, build a precise model of systems behavior (event trace notion), provide a programming formalism to describe computations over event traces, and then apply these in order to define different kinds of QoS metrics. Constructive calculations of QoS metrics on a representative set of test cases is one of cornerstones of the proposed iterative approach to system assembly from components meeting user's query specifications.

This approach to the design of a system behavior model assumes that the run time actions performed within the system may be observed as detectable events. Each event corresponding to an action is a time interval, with beginning, end, and duration. Certain attributes could be associated with the event, e.g. program state, source code fragment, time, etc. There are two binary relations defined for the event space: inclusion (one event may be nested within another), and precedence (events may be partially ordered accordingly to the semantics of the system under consideration). Hence, when executed, a system generates an event trace - set of events structured along the relations above. This event trace actually can be considered as a formal behavior model of the system ("lightweight semantics"). This model could be presented as a set of axioms about event trace structure called event grammar [1].

For example, suppose that the entire system execution is represented by an event of type execute-system. It may contain events of the type evaluate-component-A and evaluate-component-B. Event grammar may contain an axiom:
`execute-system:  (evaluate-component-A evaluate-component-B)*`
which states that `evaluate-component-A` is always followed by the `evaluate-component-B` event, and these pairs may be repeated zero or more times.

A new concept for specification and validation of target program behavior based on the ideas of event grammars and computations over program execution traces has been developed, and assertion language mechanisms, including event patterns and aggregate operations over event traces, to specify expected behavior, to describe typical bugs, and to evaluate debugging queries to search for failures (e.g. gathering run time statistics, histories of program variables, etc.) have been created. An event grammar provides a basis for QoS metrics implementation via target program automatic instrumentation. Since the instrumentation is conditional, it does not deteriorate the efficiency of the final version generated code. This mechanism based on independent models of system behavior makes it possible to define QoS metrics as generic trace computations, so that the same metric may be applied to different versions of an assembled system (via automatic instrumentation). To facilitate use of the event grammar model for the assembled system, the event definitions should be consistent through the entire component space. The QoS metrics for components should adhere to this principle. The process proposed in Section 4.4 for assembling a distributed system from components in a distributed network offers a possible approach to achieving this.

2.2.3 Infrastructure

As local autonomy is inherent in open DCS, forcing every component developer to abide by certain rigid rules, although attractive, is doomed to fail. UMM tackles the issue of non-uniformity with the assistance of the *head-hunter* and *Internet Component Broker*. These are responsible for allowing a seamless integration of different component models and sustaining a cooperation among heterogeneous (adhering to different models) components.

Head-hunter Components

The tasks of head-hunters are to detect the presence of new components in the search space, register their functionalities, and attempt at match-making between service producers and consumers. A head-hunter is analogous to a binder or a trader in other models, with one difference – a trader is passive, i.e., the onus of registration is on the foreign components and not on the trader. In contrast, a headhunter is active, i.e., it discovers other components and makes an attempt to register them with itself. There are many approaches possible for the discovery of components. They range from the standard search techniques to broadcasts and multi-casts to selected machines. At a conceptual basis, UMM does not tie itself to a specific approach but during the prototype development a particular approach will be selected for the discovery process. During registration, each component will inform the head hunter about all its aspects. The head hunter will use this information during matching. A component may be registered with multiple head-hunters. Head-hunters may cooperate with each other in order to serve a large number of components. The functionality of head hunters makes it necessary for them to communicate with components belonging to any model, implying that the cooperative aspect of head hunters be universal. Considering the heterogeneous nature of the components, it is conceivable that the software realization of a distributed system will require an ensemble of components adhering to different models. This requires a mediator, the *Internet Component Broker*, that will facilitate cooperation between heterogeneous components.

Internet Component Broker

The Internet Component Broker (ICB) acts as a mediator between two components adhering to different component models. The broker will utilize adapter technology, each adapter component providing translation capabilities for specific component architectures. Thus, a computational aspect of the adapter component will indicate the models for which it provides interoperability. It is expected that brokers will be pervasive in an Internet environment thus providing a seamless integration of disparate components. Adapter components will register with the ICB and while doing so they will indicate their specializations (which component models they can bridge efficiently). During a request from a seeker, the head hunter component will not only search for a provider, but it will also supply the necessary details of an ICB.

The adapter components achieve interoperability using the principles of *wrap* and *glue* technology [9]. A reliable, flexible and cost-effective development of wrap and glue is realized by the automatic generation of glue and wrappers based on component specifications. Wrapper software provides a common message-passing interface for components that frees developers from the error prone tasks of implementing interface and data conversion for individual components. The glue software schedules time-constrained actions and carries out the actual communication between components.

The functionality of the ICB is analogous to that of an object request broker (ORB). The ORB provides the capability to generate the glue and wrappers necessary for objects written in different programming languages to communicate transparently; the ICB provides the capability to generate the glue and wrappers necessary for components implemented in diverse component models (and providing service guarantees) to collaborate across the Internet. An ORB defines language mappings and object adapters. An ICB must provide component mappings and component model adapters. While the ICB conceptually provides the capabilities of existing bridges (COM-CORBA for example), the ICB will provide key features that are unique; it is designed to provide the auxiliary aspects of the Internet – collaboration between autonomous environments, mobility and security. In addition, the UMM includes quality of service and service guarantees. The ICB, in conjunction with head-hunters provide the infrastructure necessary for scalable, reliable, and secure collaborative business using the Internet.

3 A Case Study

In order to explain the UMM and the proposed approach, below a case study from the domain of distributed information filtering is presented. Although the case study uses a specific domain, the principles can be easily extended to other application domains that involve the software realization of a DCS.

3.1 Distributed Information Filtering

It is desired to develop a global information filtering system, in which, users will be interested in receiving selected information, based on their preferences, from scattered repositories. Usually, a filtering task involves contacting the scattered resources, performing an initial search to gather a subset of documents, representing, classifying and presenting based on the user profile. Many different methods are employed for the sub-tasks involved in filtering. Thus, it can be easily envisioned that different components, each employing a different algorithm to perform these sub-tasks, will be scattered across an interconnected system. Each component may belong to a different model, may quote different costs and offer different qualities of service.

Hence, a typical distributed information filtering system consists of the following types of components: a) Domain Component (DC), b) Wrapper Component (WC), c) Representer Component (RC), d) Classifier Component (CC), and e) User Interaction Component (UIC). In addition to these domain-specific components, headhunter components (HC) and the ICB are needed.

All these components, their aspects and characteristics need to be defined using UMM. For the sake of brevity, only the complete description of the domain component (DC) is shown below.

3.2 Domain Component

The domain component is responsible for maintaining a repository of URLs of associated information sources for particular type (e.g., text, structure, sequence) of information that needs filtering.

For example, the inherent attributes might consist of Author (name of the component developer), Version (current version of the component), Date Deployed, Execution Environment Needed and Component Model (e.g., Java-RMI 1.2.2), Validity (e.g., one month from the deployment), Atomic or Complex (indivisible or an amalgamation of other components, e.g. atomic), Registrations (with which headhunters this component is registered, e.g., H1 - `www.cs.iupui.edu/h1` and H2 - `www.cis.uab.edu/h2`).

An informal description of the functional part of a component may contain:

```
1. Computational Task Description -- e.g., searching a selected set of databases over the Internet.
2. Algorithm Used and its Complexity -- Webcrawling and O(n^2), respectively.
3. Alternative Algorithms -- Indexing.
4. Expected Resources (best, average and worst-cases) -- multi-processor, uni-processor (300MHz
with an CPU utilization of 50%), and uni-processor (100MHz with CPU utilization of 99%), respectively.
5. Design Patterns Used (if any) -- Broker.
6. Known Usages  -- for assembling an up-to-date listing containing addresses of known information
repositories for a particular domain.
7. Aliases--  such a component is usually called a Pro-active Agent.
8. Multi-level contracts:
e.g., for a function like List getURLs (Domain inputDomain, Compensation inputCost), the behavioral
contract could specify the pre-condition to be (valid Domain Name and cost), post-condition to be:
if successful (activeClientThreads++ and cost+=inputCost)
else (raise DomainNotKnownException and InvalidCostException)
and the invariant could be (ListOfURLs > 1).  Also, for the same function, the concurrency contract
could specify (maximum number of active threads allowed = 50).
```

The cooperation attributes of the domain component may consist of 1) expected collaborators UIC, WC, HC, TC and RC, 2) pre-processing collaborators HC and TC, and 3) post-processing collaborators RC and UIC.

The auxiliary attributes of the domain component are 1) fault-tolerant attributes, e.g., check-pointing versions, 2) security attributes, e.g., simple encryption, and 3) mobility attributes, e.g.. "not mobile."

For the domain component, the QoS parameters may contain 1) number of available URL's, 2) ranking of URL's, and 3) average rate of URL collection.

A component developer may offer several possible levels of QoS, e.g., L1) novice (number of URL's < 50 and no ranking of URL's and average rate of URL collection ≥ 1 week and average latency ≥ 2 minutes), L2) intermediate (number of URL's < 500 and simple ranking of URL's and average rate of URL collection ≥ 3 days and average latency ≥ 1 minute), and L3) expert (number of URL's < 1500 and advanced ranking of URL's and average rate of URL collection ≥ 1 day and average latency ≥ 5 seconds).

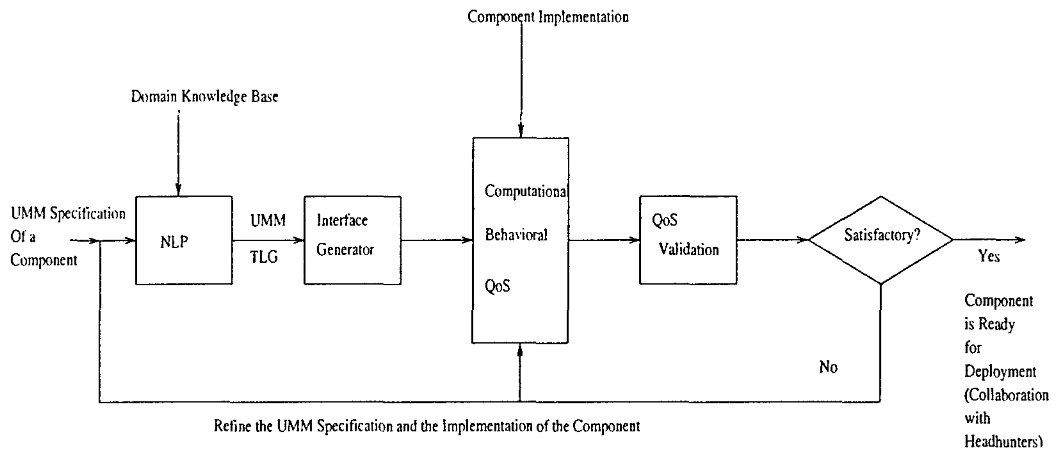

Component Implementation
Domain Knowledge Base
UMM Specification Of a Component
NLP
UMM
TLG
Interface Generator
Computational Behavioral QoS
QoS Validation
Satisfactory?
Yes
Component is Ready for Deployment (Collaboration with Headhunters)
No
Refine the UMM Specification and the Implementation of the Component

Figure 1: The Component Development and Deployment Process in UMM

The expected compensations for the above levels in terms of the number of URLs could be 1) L1 > 100 and < 200, 2) L2 > 200 and < 400, and 3) L3 > 400 and < 600.

4 Component and System Generation Using UMM Framework

The development of a software solution, using the UMM approach, for a DCS has two levels: a) component level – in this level, different components are created by developers, tested and verified from the point of view of QoS, and then deployed on the network, and b) system level – this level concentrates on assembling a collection of components, each with a specific functionality and QoS, and semi-automatically generates the software solution for the particular DCS under consideration. These two levels and associated processes are described below.

4.1 Component Development and Deployment Process

The component development and deployment process is depicted in Figure 1. As seen in the figure, this process starts with a UMM specification of a component (from a particular domain). This specification is in a natural-language format, as illustrated in the previous section. This informal specification is then refined into a formal specification. The refinement is based upon the theory of Two-Level Grammar (TLG) natural language specifications [3, 23], and is achieved by the use of conventional natural language processing techniques (e.g. see [7]) and a domain (such as information filtering) knowledge base. TLG specifications allow for the generation of the interface (possibly multi-level) for a component. This interface incorporates all the aspects of the component, as required by the UMM. The developer provides the necessary implementation for the computational, behavioral, and QoS methods. This process is followed by the QoS validation. If the results are satisfactory (as required by the QoS criteria) then the component is deployed on the network and eventually, it is discovered by one or more headhunters. If the QoS constraints are not met then the developer refines the UMM specification and/or the implementation and the cycle repeats.

4.2 Formal Specification of Components in UMM

Since the UMM specifications are informally indicated in a natural language like style, our approach is to translate this natural language specification into a more formal specification using TLG. TLG is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The name "two-level" in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars, one corresponding to a set of type declarations and the other a set of function definitions operating on those types. These type and function definitions are incorporated into a class which allows for new types to be created.

The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. On the other hand, function definitions may be given without precisely defined domains for a more flexible specification approach. This framework consists of a knowledge-base which establishes a context for the natural language text to be used in the specification under a particular domain model, in this case information filtering. This allows the TLG to be translated into internal representations such as predicate logic, the natural representation for TLG, event grammars, or multi-level Java interfaces taking the form of the UMM specification template. For the case study, we may use a TLG class to describe the component structure and functionality as elaborated in the following subsections.

4.2.1 Component Structure Specification

Syntactically, TLG type declarations are similar to those in other languages. Types are capitalized whereas constants begin with lower case letters. The usual primitive types, such as `Integer`, `Float`, `Boolean`, and `String` are present as are list constructors based upon regular expression notation, e.g. {X}* and {X}+ mean 0 or more and 1 or more occurrences of X, respectively.

The types of the domain component in our information filtering system are defined in the following way in TLG.

```
Component :: DomainComponent; WrapperComponent; RepresentationComponent; ClassificationComponent;
  UserInteractionComponent; HeadhunterComponent; ICB.
DomainComponent :: Name, InformalDescription, Attributes, Service.
Name :: dc.
Attributes :: ComputationalAttributes, CooperationAttributes, AuxiliaryAttributes.
ComputationalAttributes :: InherentAttributes, FunctionalAttributes.
InherentAttributes :: Author, Version, DateDeployed, ExecutionEnvironment,
  ComponentModel, Validity, Structure, Registrations.
FunctionalAttributes :: TaskDescription, AlgorithmAndComplexity,
  Alternatives, Resources, DesignPatterns, Usages, Aliases, FunctionsAndContracts.
AlgorithmAndComplexity :: webcrawling, n^2; ....
Alternatives :: {AlgorithmAndComplexity}*.
Resource :: Architecture, Speed, Load.
Architecture :: uni-processor; multi-processor.
Speed :: Integer.
Load :: Integer.
DesignPatterns :: broker; ....
Aliases :: pro-active agent; ....
FunctionAndContract :: Function, BehavioralContract, ConcurrencyContract.
Function :: ....
BehavioralContract :: Precondition, Invariant, Postcondition.
ConcurrencyContract :: single threaded; maximum number of active threads allowed = Integer; ....
CooperationAttributes :: ExpectedCollaborators, PreprocessingCollaborators, PostprocessingCollaborators.
ExpectedCollaborator :: uic; wc; hc; tc; rc.
PreprocessingCollaborator :: hc; tc.
PostprocessingCollaborator :: rc; uic.
AuxiliaryAttribute :: FaultTolerantAttribute; SecurityAttribute; MobilityAttribute.
FaultTolerantAttribute :: check-pointing versions; ....
SecurityAttribute :: simple encryption; ....
MobilityAttribute :: mobile; not mobile.
Service :: ExecutionRate, ParallelismConstraint, Priority, Latency, Capacity, Availability,
  OrderingConstraints, QualityOfResultsReturned, ResourcesAvailable, ....
ExecutionRate :: Float.
ParallelismConstraint :: synchronous; asynchronous.
Priority :: Integer.
Latency :: AverageRateOfURLCollection.
AverageRateOfURLCollection :: Float.
Capacity :: NumberOfAvailableURLs.
NumberOfAvailableURLs :: Integer.
Availability :: Float.
OrderingConstraint :: Boolean.
QualityOfResultsReturned :: {URL}+.
ResourcesAvailable :: HardwareResources, SoftwareResources.
HardwareResources :: ....
SoftwareResources :: ....
```

The remaining components (e.g., wrapper, representation, etc.) may be described in a similar manner. All domains not specified explicitly in the above example are assumed to be of type `String`, with the exception of `Function` which may take the form of an interface definition in a programming language such as Java. Using standard natural language processing techniques [7], the UMM specification may be automatically refined into this TLG specification, with user assistance as needed to clarify ambiguities. The process is facilitated by the presence of a knowledge base which understands the domain of information filtering from the point of view of vocabulary which may be used in making the original UMM specification.

4.2.2 Component Functionality Specification

The second level of the TLG specification is for function declarations. These resemble logical rules in logic programming with variables coming from the domains established in the type declarations. For the Domain Component example, the levels of Quality of Service may be specified as follows.

```
number of urls : size of QualityOfResultsReturned.
average latency : ...
no ranking of urls : ...
simple ranking of urls : ...
advanced ranking of urls : ...
average latency : ...
qos level 1 is novice : number of urls < 50, no ranking of urls,
  AverageRateofURLCollection >= 1 week, average latency >= 2 minutes.
qos level 2 is intermediate : number of urls < 500, simple ranking of urls,
  AverageRateofURLCollection >= 3 days, average latency >= 1 minute.
qos level 3 is expert : number of urls < 1500, advanced ranking of urls,
  AverageRateofURLCollection >= 1 day, average latency >= 5 seconds.
```

Each rule defines how the particular entity is to be computed. As these rules are normally part of a class definition encapsulating a corresponding set of type declarations, each rule has access to the data specified in the type declarations. These natural language like rules may be further refined into a more formal specification, e.g. using event grammars.

4.3 QoS Guarantee of a Domain Component

For the case study, the event grammar to describe the system behavior is given below. The first part is the set of type definitions and the second part is the description of computations over event traces implementing different QoS metrics.

```
exec_syst :: (request_sent | response_received)*
response_received :: (URL_detected | failed)
```

These type definitions describe the types of events which may occur as the system executes. The computations over these events include verification that the number of URL's detected is less than 50 and also the latency (e.g., for all requests for URL's, every response received occurs within 10 units of time). id is an event attribute which associates a unique identifier between query attributes and corresponding responses. Both of these metrics yield Boolean values.

```
CARD [URL_detected from exec_syst] < 50

Forall x : request_sent from exec_syst
  Exists y : response_received from exec_syst
    id (x) = id (y) & begin_time (y) - end_time (x) < 10
```

4.4 Automated System Generation and Evaluation based on QoS

In general, different developers will provide on the Internet a variety of possibly heterogeneous components oriented towards a specific problem domain. Once all the components necessary for implementing a specified distributed system are available, then the task is to assemble them. Figure 2 shows a process to accomplish this. The developer of the desired distributed system presents to this process a system query, in a structured form of natural language, that describes the required characteristics of the distributed system. For example, such a query might be a request to assemble an information filtering system. The natural language processor (NLP) processes the query. It does this aided by the domain knowledge (such as key concepts in the filtering domain) and a knowledge-base containing the UMM description (in the form of a TLG) of the components for that domain. The result is a formal UMM specification that will be used by headhunters for component searches and as an input to the system assembly step. This formal UMM specification will be a basis for generating a set of test cases to determine whether or not an assembly satisfies the desired QoS. The framework, with the help of the infrastructure described in Section 2.2.3, collects a set of potential components for that domain, each of which meets the QoS requirement specified by the developer. From these, the developer, or a program acting as a proxy of the developer, selects some components. These components along with the component broker and appropriate adapters (if needed) form a software implementation of the distributed system. Next this implementation is tested using event traces and the set of test cases to verify that it meets the desired QoS criteria. If it does not, it is discarded. After that, another implementation is chosen from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is found, or until the collection is exhausted. In the latter case, the process may request additional

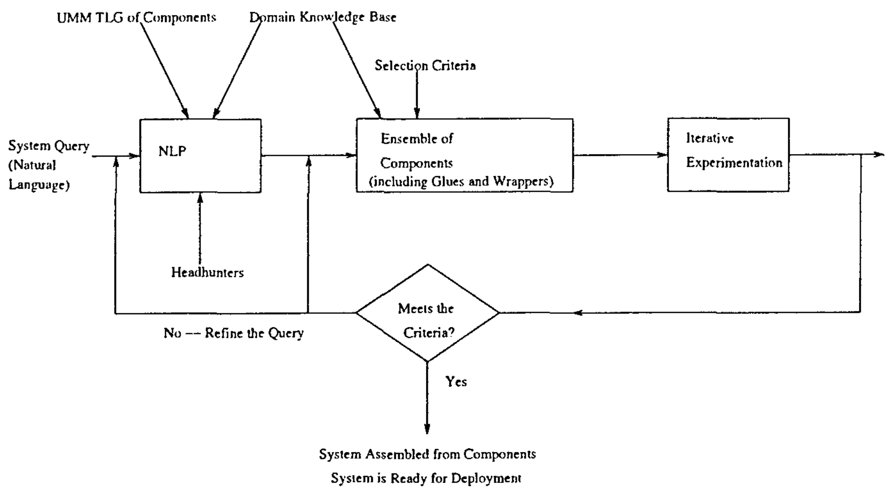

UMM TLG of Components
Domain Knowledge Base
Selection Criteria
System Query (Natural Language)
NLP
Ensemble of Components (including Glues and Wrappers)
Iterative Experimentation
Headhunters
Meets the Criteria?
No — Refine the Query
Yes
System Assembled from Components
System is Ready for Deployment

Figure 2: The Iterative System Integration Process in UMM

components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. Once a satisfactory implementation is found, it is ready for deployment.

5 Conclusion

This paper has presented a framework that allows an interoperation of heterogeneous and distributed software components. The software solutions for future DCS will require either automatic or semi-automatic integration of software components, while abiding with the QoS constraints advertised by each component and the collection of components. The result of using UMM and the associated tools is a semi-automatic construction of a distributed system. Glue and wrapper technology allows a seamless integration of heterogeneous components and the formal specification of all aspects of each component will eliminate ambiguity while detecting and using these components. The UMM does not consider network failures or other considerations related to the hardware infrastructure, however, these could be integrated into the QoS level of components. The UMM approach to validating QoS is to use event grammar to calculate QoS metrics over run-time behavior. The QoS metrics are then used as a criteria for an iterative process of assembling the resulting system as shown in Section 4.4. UMM also provides an opportunity to bridge gaps that currently exist in the standards arena. Although, the paper has only presented a case study from the domain of distributed information filtering, the principles of UMM may be applied to other distributed application domains. Future work includes refinement of the UMM feature thesaurus and methods for translating UMM specifications into Two-Level Grammar, refining the head-hunter mechanism, developing Quality of Service metrics for components and systems, and development of generation mechanisms for domain-specific component reuse.

References

[1] Auguston, M. A Language for Debugging Automation. In *Proceedings of 6th International Conference on Software Engineering and Knowledge Engineering*, pages 108–115, 1994.

[2] Beugnard, A., Jezequel, J., Plouzeau, N. and Watkins, D. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.

[3] Barrett R. Bryant. Object-Oriented Natural Language Requirements Specification. In *Proceedings of ACSC 2000, the 23rd Australasian Computer Science Conference, January 31-February 4, 2000, Canberra, Australia*, pages 24–30, January 2000.

[4] California Institute of Technology. *Caltech Infospheres On-line Documentation, URL:- http://www.infospheres.caltech.edu/*, 1998.

[5] Fox, G. The Document Object Model Universal Access Other Objects CORBA XML Jini JavaScript etc. *http://www.npac.syr.edu/users/gcf/msrcobjectsapril99*, 1999.

[6] Israel, B. and Kaiser, G. Coordinating Distributed Components Over the Internet. *IEEE Internet Computing*, pages 83-86, 2(2), 1998.

[7] Jurafsky, D. and Martin, J. H. *Speech and Language Processing.* Prentice Hall, 2000.

[8] Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S. and Cybenko, G. Agent TCL: Targetting the Needs of Mobile Computers. *IEEE Internet Computing*, pages 58–67, 1(4), 1997.

[9] Luqi, Berzins, V., Ge, J., Shing, M., Auguston, M., Bryant, B. R. and Kin, B. K. DCAPS - Architecture for Distributed Computer Aided Prototyping System. In *Proceedings of RSP 2001, the 12th International Workshop on Rapid System Prototyping*, 2001.

[10] Manola, F. Technologies for a Web Object Model. *IEEE Internet Computing*, 3(1):38–47, January-February 1999.

[11] Microsoft Corporation. *DCOM Specifications, URL:- http://www.microsoft.com/oledev/olecom*, 1998.

[12] Object Management Group. XML Metadata Interchange. Technical report, Object Management Group Document No. ad/98-10-05, October 1998.

[13] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999.

[14] Object Management Group. Meta Object Facility (MOF) Specification, Version 1.3. Technical report, Object Management Group, March 2000.

[15] Object Management Group. Model Driven Architecture: A Technical Perspective. Technical report, Object Management Group Document No. ab/2001-02-01, February 2001.

[16] Orfali R, and Harkey, D. *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, Inc., 1997.

[17] Raje, R. UMM: Unified Meta-object Model for Open Distributed Systems. In *Proceedings of the fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP'2000)*, 2000.

[18] Rogerson, D. *Inside COM*. Microsoft Press, 1996.

[19] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[20] Siegel, J. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.

[21] The Globus Project. *Globus Website, URL:- http://www.globus.org/*, 2000.

[22] University of Virginia. *Legion Project, URL:- http://www.cs.virginia.edu/ legion*, 1999.

[23] van Wijngaarden, A. Orthogonal Design and Description of a Formal Language. Technical report, Mathematisch Centrum, Amsterdam, 1965.

Two-Level Grammar as an Object-Oriented Requirements Specification Language *

Barrett R. Bryant   Beum-Seuk Lee
Department of Computer and Information Sciences
The University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL 35294-1170, U. S. A.
{bryant, leebs}@cis.uab.edu

Abstract

Two-Level Grammar (TLG) is proposed as an object-oriented requirements specification language with a natural language (NL) style but sufficiently formal to allow automatic transformation of the TLG specification into formal specifications in VDM++, an object-oriented version of the Vienna Development Method. The VDM++ specification may be further transformed into Java^{TM} code or integrated with the Unified Modeling Language (UML) using the IFAD VDM Toolbox^{TM}. The translation into an executable programming language facilitates rapid prototyping of TLG specifications and the integration with UML allows TLG specification to be used in conjunction with software systems being constructed using UML. This software specification approach is supported by a specification development environment (SDE) for constructing TLG specifications and a natural language processing system to assist in translating an NL requirements specification into TLG. The system described is a useful and constructive tool for automating the production of software systems from NL specifications.

1. Introduction

Despite a wide variety of formal specification languages [1] and modeling languages such as the Unified Modeling Language (UML) [11], natural language (NL) remains the method of choice for describing software system requirements. Informal specifications in NL must be turned into more formal designs on the way to a complete implementation. These formal requirements are necessary not only for the rapid prototyping of the evolving software systems but also to provide a standard reference model upon which all successive implementations should be constructed. Since object-oriented modeling using UML and associated tools is now a standard for software system design, there is a need for a requirements specification language which may be both conveniently used to express the original NL specification but also mapped into an object-oriented design. Since objects are already concepts in the domain of an NL vocabulary, an object-oriented design has the potential for most closely matching a requirements specification in the user's vocabulary. In fact, one technique of object-oriented analysis is to determine the objects of the problem domain using nouns in the requirements specification and determine the interactions between objects and their associated operations using verbs and their direct objects [2]. While objects may be more natural to describe in a requirements specification, some additional tools are needed to facilitate the mapping between the user's description of requirements and the actual design. Toward this end, we have developed a requirements specification language based upon Two-Level Grammar (TLG) [13] with the following advantages:

1. The NL nature of a TLG specification makes it very understandable and useful as a communication medium between users, designers, and implementors of the software system.

2. Despite an apparent NL quality, the TLG notation is sufficiently formal to allow formal specifications to be constructed using the notation.

*This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350 and by the U. S. Office of Naval Research under award number 00014-01-1-0746.

3. TLG specifications are wide-spectrum, meaning that the specification may be very detailed for implementation as well as very general for design.

4. We have developed implementation techniques to rapidly prototype the TLG specifications, when a sufficient level of detail is specified, by means of translation into efficient executable code in object-oriented programming languages.

This paper describes the details of the TLG specification language and its implementation, including type system, object-orientation, and natural language base, and shows how TLG is mapped into VDM++.

2. Two-Level Grammar

Two-level Grammar (TLG, also called W-grammar) was originally developed as a specification language for programming language syntax and semantics, and later used as an executable specification language [4], and as the basis for conversion from requirements expressed in natural language into a formal specification [3].

2.1. Language Description

The name "two-level" in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars defining the set of type domains and the set of function definitions operating on those domains, respectively. Note that while we use the term "domain" in a type-theoretic context, the notion can be scaled up to a much larger context as in domain of "objects." These grammars may be defined in the context of a class in which case the type domains define the instance variables of the class and the function definitions define the methods of the class.

2.1.1. Types. The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. In traditional TLG literature, these declarations are referred to as *meta-rules.* The function domains of TLG may be formally structured as linear data structures such as lists, sets, bags, or singleton data objects, or be configured as tree-structured data objects. The standard structured data types of product domain and sum domain may be treated as special cases of these.

Domain declarations have the following form:

```
dentifier-1, Identifier-2, ..., Identifier-m ::
 data-object-1; data-object-2; ...; data-object-n.
```

where each `data-object-i` is a combination of domain identifiers, singleton data objects, and lists of data objects, which taken together as a union form the type of `Identifier-1`, `Identifier-2`, ..., `Identifier-m`. If n=1, then the domain is a true singleton data object, whereas if n>1, then the domain is a set of the n objects. Syntactically, domain identifiers are capitalized, with underscores or additional capitalizations of successive words for readability (e.g., `IntegerList`, `Symbol_Table`, etc.), and singleton data objects are lists of NL words written entirely in lower case letters (e.g., `sorted list`). A list, set or bag structure is denoted by a regular expression or by following a domain identifier with the suffix `List`, `Set`, or `Bag`, respectively. Following conventional regular set notation, * implies a list of zero or more elements while + denotes a list of one or more elements. Furthermore, there exists a predefined environment of primitive types, such as `Integer`, `Boolean`, `Character`, `String`, etc. To clarify these, consider the following examples.

```
Person :: first name String middle initial Character
        last name String.
Persons :: PersonList.
People :: {first name String middle initial Character
        last name String}*.
Symbol_Table :: {id Identifier type Type value Integer}+.
```

`Person` denotes a product of `String`, `Character`, and `String` types, each tagged with an appropriate identifier to establish context. The types `Persons` and `People` are equivalent, as is the type `{Person}*`. `Symbol_Table` denotes a compiler symbol table configured as a list of records, each with three fields: `id`, `type` and `value`, with corresponding types `Identifier`, `Type`, and `Integer` (the first two of these are not standard TLG types and so should be explicitly declared). Each type name which appears on the right side of a declaration rule represents a value of that type, i. e., type names may be used as variables, making type declarations unnecessary although they enhance readability.

These examples have illustrated list structured types which essentially correspond to regular sets in formal language theory. Type checking then corresponds to simple pattern matching between regular sets. Determining the equivalence between two types is always decidable and checking the type of a value is equivalent to executing a deterministic finite automaton ($O(n)$).

The main difference between list structures and tree structured domains in terms of their declaration is whether the defining domain identifier declaration is recursive or not. Recursive domains are more powerful in that they allow "context-free" data types to be defined, such as expression strings with balanced parentheses as in the following example:

```
Expression :: ( Expression ).
```

The context-free grammars defining such data types may not be left recursive and must be unambiguous, so as to allow proper parsing. Left recursion is not needed since regular expression notation may be used in it's place. For example, instead of expressing:

```
Expression :: Expression + Term | Term.
```

we may express:

```
Expression :: Term {+ Term}*.
```

Type checking on tree structures corresponds to pattern matching over context-free grammars, i.e., parsing. Since we have imposed the restrictions of no left recursion and no ambiguity, we can guarantee that type checking a value may be done in $O(n)$ time using conventional context-free parsing techniques (e.g. LL (k) parsing). However, we can not determine the equivalence of two tree structured types as equivalence of context-free grammars is undecidable.

2.1.2. Functions. Function definitions comprise the operational part of a TLG specification. Their syntax allows for the semantics of the function to be expressed using a structured form of natural language. In traditional TLG literature, these are referred to as *hyper-rules.* Function definitions take the forms:

```
function signature.
function signature :
  FunctionCall-1, FunctionCall-2, ..., FunctionCall-n.
```

where n$\geq$1. Function signatures are a combination of NL words and domain identifiers, corresponding to variables in a logic program. Some of these variables will typically be input variables and some will be output variables, whose values are instantiated at the conclusion of the function call. Therefore, functions usually return values through the output variables rather than directly, in which case the direct return value is considered as a Boolean `true` or `false`. `true` means that control may pass to the next function call, while `false` means the rule has failed and an alternative rule should be tried if possible. Alternative rules have the same format as that given above. If multiple function rules have the same signature, then the multiple left hand sides may be combined with a ; separator, as in:

```
function signature :
  FunctionCall-11, FunctionCall-12, ..., FunctionCall-1j;
  FunctionCall-21, FunctionCall-22, ..., FunctionCall-2k;
  ...
  FunctionCall-n1, FunctionCall-n2, ..., FunctionCall-nm.
```

where there are n alternatives, each having a varying number of function calls. Besides Boolean values, functions may return regular values, usually the result of arithmetic calculations. In this case, only the last function call in a series should return such a value.

An important aspect about TLG is that the functions may be written at a very high level of abstraction (e.g. `compute the total mass and total cost`) or embedded into a domain definition as in traditional object-oriented programs (e.g. `compute the TotalMass and TotalCost of This Part by computing the TotalMass and TotalCost of its Subparts`, which might be embedded as a method in a `Part` class). The use of NL in the function may be regarded as a form of infix notation for functions, in contrast with the customary prefix forms of most other programming languages. It is similar to multi-argument message selectors in Smalltalk but provides even greater flexibility, including the presence of logical variables, denoted by the use of domain names (capitalized). This notation provides a highly readable way of writing what is to be done and is wide-spectrum in the sense that "what is to be done" may be expressed at multiple levels. The functions typically return a Boolean value as the main operation is to instantiate the logical variables, but simple function values such as arithmetic expressions may also be computed. These function definitions form the basis for the initial design. In an implementation, they may be represented by functions in traditional object-oriented programming languages, such as Java.

A function may be defined as a *rule.* For example, we could define an expensive part using the syntax `Expensive part : part with an imported base part and cost more than $100.` or alternatively we could write in more natural form `Expensive parts are parts with an imported base part and cost more than $100.` An implementation would transform the second form into the first, and even that form into the more formal rule for `Part` objects: `expensive : BasePart imported, Cost > 100.`

To explain the operational semantics of TLG function rules, note that each function call on the right hand side of a function definition should correspond to a function signature defined within the scope of the TLG program or be a special operation such as a Boolean comparison, assignment statement, or if-then-else statement. Every domain identifier with the same name is instantiated to the same value within a function invocation. This is called *consistent substitution.* If variables have the same root name but are numbered, then the numbers are used to distinguish between variables. A numbered variable `V1` will then be different from a variable `V2` and the two can have different values. However, they will be of the same type, namely type `V`. Once a variable has been assigned a value, it may not be reassigned, unless it is an instance variable of a class, and even in this case, it would not be usual to do so in the same function. Each function definition may therefore be thought of as a set of logical rules. The function calls are executed in the order given in the function definition. Functions may be recursive with the expected operational behavior.

Besides defined functions, TLG supports the usual arithmetic and Boolean operations, as well as list comprehensions and iterators over lists. The syntax of a list comprehension is `list all Element from ElementList1 such that Element condition giving ElementList2`. This returns a list, `ElementList2`, of all `Element` values in `ElementList` satisfying the given condition. The syntax of an iterator is `select Element from ElementList with Element condition`. This returns the first `Element` from `ElementList` which satisfies the condition.

To explain the language further, consider the following examples.

Example 1. Palindrome.

```
Character is a palindrome.
Character String Character is a palindrome :
  String is a palindrome.
```

This TLG specification has no explicit type declarations since the function rules use the type names directly as variables. The two function rules are mutually exclusive, the first handling single characters and the second handling strings of two or more characters. The second rule matches if and only if the first and last characters of the string argument are the same.

Example 2. Quick Sort.

```
Pivot :: Integer.
IntegersLess, IntegersGreater, SortedIntegersLess,
  SortedIntegersGreater :: IntegerList.

quick sort Empty into Empty.

quick sort Pivot IntegerList into SortedIntegersLess
    Pivot SortedIntegersGreater :
  split IntegerList into lists IntegersLess and
    IntegersGreater using Pivot,
  quick sort IntegersLess into SortedIntegersLess,
  quick sort IntegersGreater into SortedIntegersGreater.

split Empty into lists Empty and Empty using Pivot.

split Integer IntegerList into lists Integer IntegersLess
    and IntegersGreater using Pivot :
  Integer <= Pivot,
  split IntegerList into lists IntegersLess and
    IntegersGreater using Pivot.

split Integer IntegerList into lists IntegersLess and
    Integer IntegersGreater using Pivot:
  Integer > Pivot,
  split IntegerList into lists IntegersLess and
    IntegersGreater using Pivot.
```

The two `quick sort` rules are mutually exclusive, but the second and third `split` rules may both match non-empty lists. Each of these two `split` rules serves to distribute the `Integer` at the beginning of the list to the `IntegersLess` list or `IntegersGreater` list, depending on its relationship to `Pivot`. The first function call in each case serves as a guard to distinguish the two rules. This could have been written using an if-then-else construction, avoiding the need for the guard.

```
split Integer IntegerList into lists IntegerList1 and
    IntegerList2 using Pivot :
  split IntegerList into lists IntegersLess and
    IntegersGreater using Pivot,
  if Integer <= Pivot then begin
    IntegerList1 := Integer IntegersLess,
    IntegerList2 := IntegersGreater,
  end
  else begin
    IntegerList1 := IntegersLess,
    IntegerList2 := Integer IntegersGreater,
  end.
```

This imperative style of writing TLG's includes the begin-end grouping block and assignment statements.

The `split` rule may be eliminated completely by using list comprehensions to determine the `IntegersLess` and `IntegersGreater`, as shown below.

```
quick sort Pivot IntegerList into SortedIntegersLess
    Pivot SortedIntegersGreater :
  list all Integer from IntegerList such that
    Integer <= Pivot giving IntegersLess,
  quick sort IntegersLess into SortedIntegersLess,
  list all Integer from IntegerList such that
    Integer > Pivot giving IntegersGreater,
  quick sort IntegersGreater into SortedIntegersGreater.
```

Note that the variable `Integer` appearing in the `list all` function is not actually instantiated and so may be used in both `list all` functions without confusion.

2.1.3. Classes. TLG domain declarations and associated functions may be structured into a class hierarchy supporting multiple inheritance. The syntax of TLG class definitions is:

```
class Identifier-1
  [extends Identifier-2, ..., Identifier-n].
  {instance variable and method declarations}
end class [Identifier-1].
```

`Identifier-1` is declared to be a class which inherits from classes `Identifier-2`, ..., `Identifier-n`. In the above syntax, square brackets are used to indicate the extends clause is optional so a class need not inherit from any other class. The instance variables comprising the class definition are declared using the domain declarations described earlier. In general, the scope of these domain declarations is limited to the class in which they are defined, while the methods, corresponding to TLG function definitions, have scope anywhere an object of the given class is referred to. These notions of scoping correspond to *private* and *public* access respectively in object-oriented languages such as Java, and either scope may be declared explicitly or the scope may be made *protected*. Methods are called by writing a sentence or phrase containing the object. The result of the method call is to instantiate the logical variables occurring in the method definition.

For every class, there are predefined methods beginning with `This` which serve only to select the instance variables of a class (e.g., `This InstanceVariable` returns the value of `InstanceVariable`). `This` serves as a special variable used within the method body to denote the object to which the method is being applied. Likewise, for every instance variable of simple type there are `get` and `set` methods to access or modify that variable. For every instance variable of list type, there are `add` and `remove` methods. These are assumed and do not need to be explicitly defined.

TLG class declarations serve to encapsulate the TLG domain declarations and function definitions. The class hierarchy which is resident in TLG is a small forest of built-in classes, such as integers, lists, etc. The "main" program is nothing more than a set of object declarations using the existing class identifiers as domain names and a "query" of the appropriate methods.

3. Implementation

To effectively use TLG in the requirements specification process, we have developed a Specification Development Environment (SDE) which facilitates the construction of TLG specifications from requirements documents expressed in natural language, and then translates TLG specifications into executable code. NL requirements are translated into TLG through Contextual Natural Language Processing (CNLP) [10] which constructs a knowledge representation of the requirements which may be expressed using TLG. The TLG is then translated into VDM++ [5], the object-oriented extension of the Vienna Development Method (VDM) Specification Language (VDM-SL) [9]. The IFAD VDM Toolbox [8] is then used to generate code in an object-oriented programming language such as Java. The complete system structure is shown in Figure 1.

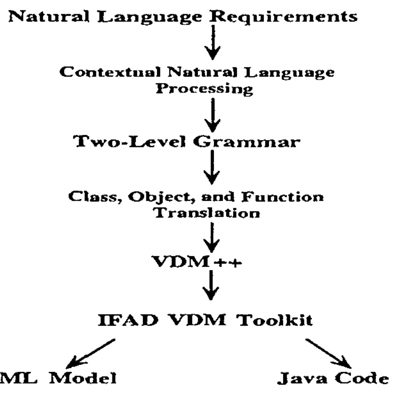

Natural Language Requirements
Contextual Natural Language Processing
Two-Level Grammar
Class, Object, and Function Translation
VDM++
IFAD VDM Toolkit
UML Model
Java Code

Figure 1. Structure of Specification Development Environment

These components are explained in the following sections in terms of an example, the Automatic Teller Machine (ATM) requirements specification below.

```
The bank keeps the list of accounts.
Each account has three integer data fields; ID, PIN, and
balance. The ATM machine has 3 service types; withdraw,
deposit, and balance check. For each service first it
verifies ID and PIN from the bank.

Withdraw service withdraws an amount from the account of
ID with PIN in the bank in the following sequence:
First it gets the balance of the account of ID from
the bank, if the amount is less than or equal to the
balance then it decreases the balance by Amount,
updates the balance of the account of ID in the bank,
and then outputs the new balance.

Deposit service deposits an amount to the account of ID
with PIN in the bank in the following sequence:
First it gets the balance of the account of ID from the
bank, it increases the balance by Amount, updates the
balance of the account of ID in the bank, and then
outputs the new balance.

Balance check service checks the balance of the account
of ID with PIN in Bank in the following order:
It gets the balance of the account of ID from the bank,
and then outputs the balance.

Transfer service withdraws an amount from the account of
ID1 with PIN in the bank and deposits the amount to the
account of ID2.
```

3.1. Processing NL Requirements Specifications

The SDE has NL parsing capabilities as well as a lexicon to aid in classification of words into nouns (objects) and verbs (operations on objects) and their relationship. Since all domain knowledge is specified by the domain definitions of the specification, the requirements written by the user can be parsed to determine the object being acted upon and the operation needed to be performed. This initial analysis of the requirements document provides the basis for further refinement according to the syntax of Two-Level Grammar function and domain definitions. The SDE analyzes each function definition and attempts to classify from the NL text which domains were involved, including the primary domain, perhaps a class, the function belongs to. A sufficient degree of interaction with the user ensures a correct interpretation. Any aspect of the specification which cannot be understood by the system can be resolved through further querying of the user. This may include the specification of additional domains and/or functions which make the specification more detailed. Once the system has "understood" the requirements that the user has specified, it can proceed with the transformation into the design and the underlying design tool can further refine this into a prototype implementation for the user to review. This process may be repeated iteratively until the requirements have been sufficiently developed to satisfy both the user and designer. By "user" we refer to either the end-user who has commissioned the system or requirements specification engineer working with the end-user. The designer can then finalize the mapping of the requirements specification into the final design. Applying this NL processing front end to the ATM requirements specification gives the following TLG.

```
class Account.
  Id, Pin, Balance, Amount :: Integer;

  withdraw Amount giving Balance1 :
    Amount <= Balance,
    Balance1 := Balance - Amount,
    set balance to Balance1.

  deposit Amount giving Balance1 :
    Balance1 := Balance + Amount,
    set balance to Balance1.
end class.

class Bank.
  Accounts :: AccountList.
  Id, Pin :: Integer.

  get account using Id giving Account :
    select Account from Accounts
      with id of Account = Id.

  get account using Id and Pin giving Account :
    select Account from Accounts with
    id of Account = Id and pin of Account = Pin.
end class.

class ATM.
  Id, Pin, Balance, Amount :: Integer.

  withdraw Amount from account of Id with Pin in Bank
      giving Balance :
    get account from Bank using Id and Pin
      giving Account,
    withdraw Amount from Account giving Balance.

  deposit Amount account of Id with Pin in Bank
      giving Balance :
    get account of Bank using Id and Pin
      giving Account,
    deposit Amount to Account giving Balance.

  check balance of Id with Pin in Bank giving Balance :
    get account of Bank using Id and Pin giving Account,
    get balance of Account giving Balance.

  transfer Amount from account of Id1 with Pin1 to
      account of Id2 in Bank :
    withdraw Amount from account of Id1
      with Pin1 in Bank giving Balance1,
    get account of Bank using Id2 giving Account2,
    deposit Amount to Account2 giving Balance.
end class.
```

It can be seen that the TLG is a structured form of the original NL specification. The exact same vocabulary is used as it is extracted by the NL processing front end. Additional information is added as needed to provide object data member access, e.g., get functions to access component objects.

Previous work in the area of NL specification of requirements includes a software reuse system which uses NL descriptions of library components to facilitate their selection for incorporation into an implementation [7], and "controlled natural language" [6], which is NL of a specific syntax with all vocabulary coming from a fixed domain. The latter system is able to translate the controlled NL specifications into Prolog so that they may be executed. We believe that our object-oriented approach to this problem offers a number of advantages with respect to both formal specification and object-oriented modeling.

3.2. Translation of TLG into VDM++

VDM++ has been selected as the target specification language for TLG because VDM++ has many similarities in structure to TLG and also has tool support for analysis and code generation. Although TLG and VDM++ are both formal specification languages, the translation from TLG into VDM++ is not simply a direct mapping between them. We will first give an overview of VDM++ and then explain how TLG is translated into VDM++.

3.2.1. VDM++. The structure of a VDM++ specification is organized as a collection of classes which take the following general form:

```
class identifier
```

```
  [is subclass of identifier-1, ..., identifier-n]
  value definitions
  type definitions
  instance variable definitions
  operation definitions
end identifier
```

Value and type definitions define constants and types that may be used in the class, respectively. VDM++ types include the basic data types as well as compound types in the form of sets, sequences, and maps. Instance variable definitions are the state variables of the class. Operation definitions correspond to methods. Operations have a signature and a body which may be an expression in the style of functional programming languages or a collection of imperative statements with `return` statements to return the function values. VDM++ also includes the option of defining state invariants, and pre-conditions and post-conditions for operations. Synchronization of concurrent operations and multi-threading are also provided for. At present we do not use these features in our translation schemes.

3.2.2. Translation Schemes. The translation of class definitions, including with inheritance, and compound type declarations, may be described through the tables shown in Figures 2 and 3. The translation of basic types is straightforward and so is not shown here. Type declarations in TLG specifications occur in class definitions for two purposes: 1) to define an instance variable of the class, and 2) to define variables which may be used in function definitions, either as function arguments or to calculate intermediate values. These are not difficult to distinguish as instance variables are related only to the state of the object and so must be used in function definitions other than as function arguments, typically a `get` or `set` operation. It is also straightforward to determine a variable used only for intermediate value calculation as such a variable will always be written before it is read - instance variables must have some function which reads them only.

A TLG function is translated into a VDM++ operation. TLG variables local to that function will be translated into VDM++ function local variables. Figure 4 indicates the general scheme for function definitions, which essentially consist of a function signature and a series of function calls. In these translations schemes, `Arg-1`, `Arg-2`, etc., are the arguments to the function, `Return-1`, `Return-2`, etc., are the results of the function, and `Arg-Type-i` and `Return-Type-i` are their respective types. The declaration of a result variable occurs only if the variable is not an instance variable of the class. This would not normally be the case unless the function was a get method associated with that instance variable. Since TLG functions may return many result values whereas VDM++ operations only return a single value, these multiple result values should be constructed into a product for the purpose of returning them as a single value. The `mk_` operation accomplishes this. `mk_` is not needed if only one return value is required. Figure 4 also shows the translation schemes for function calls. The declaration of a return variable occurs only if the variable has not been declared previously either as a return variable of the function definition in which the function call appears, or as an instance variable of the class. Since function g may return multiple values, the VDM++ operation returns a product of those values which may then be extracted into the individual values.

In addition to returning the values of result variables, TLG functions will either succeed or fail, as in logic programming predicates. Failure implies that no result variables are instantiated. This situation must be detected by VDM++ operations corresponding to those functions. In our generated VDM++ code, a special Boolean variable is introduced into the state of every object to indicate whether an operation performed on that object succeeded or failed. If the operation O fails, then so does the operation O' that invoked O, the operation that invoked O', etc. That is, this failure may be propagated to each previous operation until it causes the entire operation to fail or an alternative operation is possible. An alternative operation is one in which multiple rules are given for the same function signature. For function definitions defined by several rules, TLG uses pattern matching to determine which rule is appropriate. This pattern matching is implemented in VDM++ by either comparisons in cases where the pattern is a simple data type or by VDM++ pattern matching for compound data types. The examples in Figures 5 and 6 illustrate each case. Note that the `factorial` function is not defined over all integers as the TLG rules will succeed only for natural numbers. Therefore, the VDM++ operation may fail on a negative number argument, rendering the return value invalid. Functions calling `factorial` must also check for this failure. This does not include the recursive call since it can be detected that `factorial (n - 1)` will never fail since `n > 1`.

3.2.3. Example. The VDM++ translation of our running example, according to the rules given in the previous section, is shown below. As with the generated TLG, this code has been distilled for readability.

```
class Account
  instance variables
    id, pin, balance : int;

  operations
```

Simple Class

TLG	VDM++

class C.
 	
domain declarations
 	
function definitions
end class.
	class C
	instance variables
	variable declarations
	operations
	operation definitions
	end C

Class With Inheritance

TLG	VDM++

class SC
extends C.
. . .
end class.
	class SC
	is subclass of C
	. . .
	end C

Figure 2. Translation Schemes for Classes

TLG	VDM++	Type
DataObj :: DataTypeSet.	DataObj = set of DataType	Set
DataObj :: DataTypeList.	DataObj = seq of DataType	Sequence
DataObj :: {DataType}*.	DataObj = seq of DataType	Sequence
DataObj :: {DataType}+.	DataObj = seq1 of DataType	Sequence
DataObj :: DataType1 DataType2.	DataObj = DataType1 * DataType2	Product
DataObj :: {DataName1 DataType1 DataName2 DataType2}.	DataObj = DataName1 : DataType1 DataName2 : DataType2	Composite
DataObj :: DataType1; DataType2.	DataObj = DataType1 | DataType2	Union

Figure 3. Translation Schemes for Compound Data Types

Function Definitions

TLG
f of Arg-1 and ... and Arg-n giving Return-1 and ... and Return-m : function calls

VDM++
f : ArgType-1 * ... * ArgType-n ==> ReturnType-1 * ... * ReturnType-m f (arg-1, ..., arg-n) == (dcl Return-1 : ReturnType-1; ... dcl Return-m : ReturnType-m; function calls return mk_ (Return-1, ..., Return-m))

Function Calls

TLG
g of Arg-1 and ... and Arg-n giving Return-1 and ... and Return-m

VDM++

dcl Return-1 : ReturnType-1;
...
dcl Return-m : ReturnType-m;
dcl Returns :
ReturnType-1 * ... ReturnType-m;
Returns := g (Arg-1, ..., Arg-n);
Return-1 := Returns . #1;
...
Return-m := Returns . #m;

Figure 4. Translation Scheme for Functions

TLG	VDM++
factorial of 0 : 1. factorial of Integer : Integer > 1, Integer * factorial of (Integer - 1).	factorial : int ==> int factorial (n) == if n = 0 then return 1 elseif n > 1 then return n * factorial (n - 1) else (fail := true; return 0)

Figure 5. Simple Data Type Pattern Matching

TLG

```
quick sort Empty into Empty.
quick sort Pivot IntegerList into SortedIntegersLess Pivot SortedIntegersGreater :
   split IntegerList into lists IntegersLess and IntegersGreater using Pivot,
   quick sort IntegersLess into SortedIntegersLess,
   quick sort IntegersGreater into SortedIntegersGreater.
```

VDM++

```
quicksort : seq of int ==> seq of int
quicksort (pivotIntegerList) ==
   cases pivotIntegerList :
      [] -> return [];
      [pivot] ^ integerList ->
         (dcl splitReturns, integersLess, integersGreater : seq of int;
          dcl sortedIntegersLess, sortedIntegersGreater : seq of int;
          splitReturns := split (integerList, pivot);
          integersLess := splitReturns . #1; integersGreater := splitReturns . #2;
          sortedIntegersLess := quicksort (integersLess);
          sortedIntegersGreater := quicksort (integersGreater);
          return sortedIntegersLess ^ [pivot] ^ sortedIntegersGreater
         )
   end
```

Figure 6. Compound Data Type Pattern Matching

```
    ... getId, setId, getPin, setPin, etc. ...

    withdraw : int ==> int
    withdraw (amount) ==
      (dcl amount : int;
       if amount <= balance then
         (dcl balance1 : int;
          balance1 := balance - amount;
          setBalance (balance1)
         );
       return balance
      );

    deposit : int ==> int
    deposit (amount) ==
      (dcl amount, balance1 : int;
       balance1 := balance + amount;
       setBalance (balance1);
       return balance
      );
end Account

class Bank
  instance variables
    accounts : seq of Account;

  operations
    .. addAccount and removeAccount ...

    getAccountById : int ==> Account
    getAccountById (id) == ...

    getAccountByIdPin : int * int ==> Account
    getAccountByIdPin (id, pin) == ...
end Bank
```

```
class ATM
  instance variables
    bank : Bank;

  operations
    ... getBank and setBank ...

    withdraw : int * int * int ==> int
    withdraw (amount, id, pin) ==
      (dcl account : Account;
       dcl balance : int;
       account := bank . getAccountByIdPin (id, pin);
       balance := account . withdraw (amount);
       return balance
      );

    deposit : int * int * int ==> int
    deposit (amount, id, pin) ==
      (dcl account : Account;
       dcl balance : int;
       account := bank . getAccountByIdPin (id, pin);
       balance := account . deposit (amount);
       return balance
      );

    checkBalance : int * int ==> int
    checkBalance (id, pin) ==
      (dcl account : Account;
       dcl balance : int;
       account := bank . getAccountByIdPin (id, pin);
       balance := account . getBalance ();
       return balance
      );
    transfer : int * int * int * int ==> ()
    transfer (amount, id1, pin1, id2) ==
      (dcl account2 : Account;
       dcl balance, balance1 : int;
       balance1 := withdraw (amount, id1, pin1);
       account2 := bank . getAccountById (id2);
       balance := account2 . deposit (amount)
       return;
      );
end ATM
```

4. Summary and Conclusions

Two-Level Grammar has been presented as an object-oriented requirements specification language which is natural language-like in style but sufficiently formal to allow automatic transformation of the TLG specification into a VDM++ object-oriented formal specification. The IFAD VDM Toolbox provides for an integration of VDM++ with the Unified Modeling Language (UML) [11] through a link between the Rational Rose 2000^{TM} [12] implementation of UML and VDM++. This tool translates between UML and VDM++ and so supports round-trip engineering which may be iterative. Presently we use this in a single direction, from TLG to VDM++ to UML. This effectively allows for UML modeling of the TLG specification and so is useful for integration with existing UML models. Rational Rose does provide an "Add-In" mechanism with which we hope to have a direct integration with TLG in the future. The translation into an executable programming language using the IFAD VDM++ to Java code generator facilitates rapid prototyping of TLG specifications. Our approach to software specification is supported by a specification development environment (SDE) for constructing TLG specifications and a natural language processing system to assist in translating an NL requirements specification into the TLG. The system is a useful and constructive tool for automating the production of software systems from NL specifications.

At present the SDE exists only in prototype form but is able to handle simple NL specifications, as our example illustrated. We are extending this system so that more complex NL specifications may be handled. We would also like to automate the interaction between our SDE and tools like Rational Rose directly, in addition to going through VDM++. This will give us a complete visual modeling tool not only for object-oriented design but also for specification as well.

Acknowledgements. The authors would like to thank IFAD for providing an academic license to the IFAD VDM Toolbox in order to conduct this research.

References

[1] V. S. Alagar and K. Periyasamy. *Specification of Software Systems.* Springer-Verlag, 1998.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[3] B. R. Bryant. Object-Oriented Natural Language Requirements Specification. *Proc. ACSC 2000, 23rd Australasian Computer Science Conf.*, pages 24–30, 2000.

[4] B. R. Bryant and A. Pan. Formal Specification of Software Systems Using Two-Level Grammar. *Proc. COMPSAC '91, 15th Ann. Intl. Computer Software and Applications Conf.*, pages 155–160, 1991.

[5] E. H. Dürr and J. van Katwijk. VDM++ - A Formal Specification Language for Object-Oriented Designs. *Proc. TOOLS USA '92, 1992 Technology of Object-Oriented Languages and Systems USA Conf.*, pages 63–278, 1992.

[6] N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). *Proc. CLAW '96, First Intl. Workshop Controlled Language Applications*, 1996.

[7] M. Girardi and B. Ibrahim. A Software Reuse System Based on Natural Language Specifications. *Proc. ICCI '93, 5th Intl. Conf. Computing and Information*, pages 507–511, 1993.

[8] IFAD. The VDM++ Toolbox User Manual. Technical report, IFAD (`http://www.ifad.dk`), 2000.

[9] P. G. Larsen, et al. Vienna Development Method - Specification Language - Part I: Base Language. Report, International Standard ISO/IEC 13817-1, December 1996.

[10] J. McCarthy. Notes on Formalizing Context. Technical report, Computer Science Department, Stanford University, Stanford, CA, 1993.

[11] Object Management Group. OMG Unified Modeling Language Specification, Version 1.3. Technical report, Object Management Group, June 1999.

[12] T. Quatrani. *Visual Modeling with Rational Rose 2000 and UML.* Addison-Wesley, 2000.

[13] A. van Wijngaarden. Orthogonal Design and Description of a Formal Language. Technical report, Mathematisch Centrum, Amsterdam, 1965.

(Appeared in *Concurrency and Computation: Practice and Experience, 2002; 14:1009-1034*)

A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components

Rajeev R. Raje[1] Barrett R. Bryant [2] Andrew M. Olson[1] Mikhail Auguston[3] Carol Burt[2]

Abstract

Component-based software development offers a promising solution for taming the complexity found in today's distributed applications. Today's and future distributed software systems will certainly require combining heterogeneous software components that are geographically dispersed. For the successful deployment of such a software system, it is necessary that its realization, based on assembling heterogeneous components, not only meets the functional requirements, but also satisfies the non-functional criteria such as the desired QoS (quality of service). In this paper, a framework based on the notions of a meta-component model, a generative domain model and QoS parameters is described. A formal specification based on Two-Level Grammar is used to represent these notions in a tightly integrated way so that QoS becomes a part of the generative domain model. A simple case study is described in the context of this framework.

Keywords: Distributed systems, Quality of Service, Generative Domain Models, Heterogeneous Components, Formal methods, Two-Level Grammar.

1 Introduction

In the recent past, component-based software design has emerged as a viable and economical alternative to the traditional software design process. The notion of independently created and deployed components, with public interfaces and private implementations, loosely integrating with one another to realize a software solution is appealing. It is even more so in the field of distributed computing, where the underlying heterogeneity can be masked by the use of a coalition of distributed software components. Due to the inherent complexities of the distributed computing paradigm and due to the nascent nature of the component-based approach in this context, the potential of this approach has yet to be fully exploited. Many challenging issues need to be addressed in order to fully harness the potential of the component-based approach to distributed systems. The prominent ones are: a) the creation of a formal meta-component model, b) a mechanism to precisely describe the meta-model and associated features , c) the formalization of QoS (Quality of Service) offered by components, and d) a mechanism to assure the specified QoS. Thus, a comprehensive framework that will encompass these issues and aid the software developers is needed. In this paper, one such framework (called UniFrame) is proposed and applied to a case study.

The rest of the paper is organized as follows. The next section contains a brief description of the related efforts. It is followed by the details of Unified Meta-component Model, which accomplishes (a) and (b) above, and a brief discussion of the Generative Domain Model (GDM), which provides the domain knowledge necessary to support semi-automatic generation of component-based systems. The part of the

[1]Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 723 W. Michigan Street, SL 280, Indianapolis, IN 46202, USA, {rraje, aolson}@cs.iupui.edu, +1 317 274 5174/9733

[2]Department of Computer and Information Sciences, The University of Alabama at Birmingham, 1300 University Blvd., Birmingham, Alabama 35294-1170, USA, {bryant, cburt}@cis.uab.edu, +1 205 934 2213

[3]Department of Computer Science, New Mexico State University, Las Cruces, New Mexico 88003, USA, mikau@cs.nmsu.edu, +1 505 646 5286

framework that deals with Quality of Service (QoS) is described in section 4. Section 5 discusses Two-level Grammar (TLG), which is a formal specification method that provides a unified way to express natural language requirements, including QoS, and facilitates the semi-automatic generation of systems that satisfy them. Section 6 describes a simple case study. It illustrates the UMM description of components in a query to assemble a system from them, the TLG specifications for such components, and how to assemble them into the requested system. It does not illustrate how the GDM represents a family of architectural design specifications from which a TLG selects one appropriate for a given query. A short conclusion summarizes the observations of the paper.

2 Related Work

2.1 Component Models

Several communities have provided component (and/or collaboration) models, interoperability protocols, and directory services. These include the Object Management Group (OMG), World Wide Web Consortium (W3C), Universal Description, Discovery, and Integration (UDDI), Java Community Process (JCP), and Organization for the Advancement of Structured Information Standards (OASIS). The work of these organizations and its relevance to this research is being monitored and reported in UniFrame publications such as [11, 10]. The OMG's CORBA Component Model (CCM^{TM}) [30] provides th specification for a component framework (compatible with J2EE^{TM} - Java 2 Enterprise Edition) that enables th deployment of containers for non-Java components that can interoperate with Enterprise Java Beans (EJB). The new Model Driven Architecture (MDA) initiative is the way that the OMG will begin to standardize Platform Independent Model (PIMs) that can be mapped to multiple Platform Specific Models (such as CORBA, J2EE, Component Object Model (COM^{TM}) [35], .NET [28], and/or Web Services) for implementation. This approach holds promise for the standardization of components that could potentially be used in collaborative environments as a result of their common semantic model. W3C has progressed from pure information exchange to defining a messaging protocol using W3C standards (SOAP) and a service definition language (WSDL), which form the foundation of the "Web Services" Architecture. However, W3C has not published a component architecture.

There are also significant other research projects in this area; such as [29, 32], Web-component model/DOM [25], Pragmatic component web [14], Hadas [19], Infospheres [12], Legion [40], and Globus [16]. Some of these are language-centric, while others allow a limited interoperability. Some are general-purpose, i.e., not concentrating on any particular application domain, while others are domain-dependent. However, almost all of these models do not assume the presence of others. Thus, the interoperability which they provide is limited mainly to the underlying hardware, operating system and/or implementation languages. If component-based distributed software systems are to become successful, then there is certainly a need for an approach that will transcend this limited interoperability. One possible approach to achieve comprehensive interoperability is that of using a meta-model for heterogeneous distributed components.

2.2 Generative Programming

In [13] the generative programming paradigm is defined as: "Generative Programming is about manufacturing software products out of components in an automated way. It requires two steps: a) a design and implementation of a generative domain model, representing a family of software systems (development for reuse), this model includes also domain-specific software generator; b) given a particular requirements specification, a highly customized and optimized end-product can be automatically manufactured from implementation components by means of generation rules (development with reuse)". The notion of generative programming is incorporated in the proposed approach as described in section 3.2.2.

2.3 Quality of Service (QoS)

Although QoS and its guarantees have been widely used in networking, not many attempts have been made to incorporate QoS into component-based software systems. Quality Objects (QuO) [5] is a framework for providing QoS to software applications composed of objects (especially CORBA-based objects) that are distributed over wide area networks. QuO bridges the gap between the socket-level QoS and the distributed object level QoS. QuO's emphasis is on specification, measuring, controlling, and adapting to changes in QoS. RAPIDware [27] is an approach for component-based development of adaptable and dependable middleware. It uses rigorous software development methods to support interactive applications executed across heterogeneous networked environments. It focuses on specification, design, and use of component-based middleware.

3 Unified Meta-Component Model and Generative Domain Model

3.1 Why a Meta-model?

Given the plethora of component-based models and noting the fact that components, by definition, are independent of the implementation language, tools and the execution environment, it is necessary to answer the questions: *why is a meta-model needed for a seamless interoperation of distributed heterogeneous components?* and *how would a meta-model assist in seamlessly integrating distributed heterogeneous software components?* The answer to these question lies in: a) in any organization, software systems undergo changes and evolutions, b) local autonomy is an inherent characteristic of today's geographically (or logically) dispersed organizations, and c) if reliable software needs to be created for a distributed computing system (DCS) by combining components, then the QoS offered by each component needs to become a central theme of the software development approach.

The consequence of constant evolutions and changes is that there is a need to create prototypes rapidly and experiment with them in an iterative manner. Thus, there is no alternative but to adhere to cyclic (manual or semi-automatic) component-based software development for a DCS. However, the solution of decreeing a common COTS environment, in an organization, is against the principle of local autonomy. Hence, the development of a DCS in an organization will, most certainly, require creating an ensemble of heterogeneous components, each adhering to some model. Also, every DCS is designed and developed with a certain goal in mind, and usually that goal is associated with a certain perception of the quality (as expected from the system) and related constraints.

Thus, there is a need for a comprehensive meta-model that will seamlessly encompass existing (and future) heterogeneous components by capturing their necessary aspects, including the quality of service and associated guarantees offered components. As distributed systems are becoming omni-present and many of them are mission-critical, their software development should emphasize and integrate the QoS-oriented theme.

For enterprise component solutions, the standards necessary to design systems using a meta-model that can be realized in many diverse technologies is an area where significant standards work is now focused. The recent shift in focus for the OMG to "Model Driven Architecture" (MDA) [31] is a recognition that to create mechanized software for the collaboration and bridging of component architectures will require standardization not only of infrastructure but also Business and Component Meta-Models. The need to support the evolution of component models and to describe the capabilities of the models will be key to realizing the full potential of an E-business economy.

3.2 Unified Meta-component Model (UMM) and Unified Approach (UA)

In [33, 34] a unified meta-component model (UMM) and a unified approach (UA) based on it, for distributed component-based systems, are proposed. A brief description of UMM and UA is presented below. A more detailed discussion of UMM and UA is found in [33, 34].

3.2.1 UMM

The core parts of the UMM are: *components*, *service and service guarantees*, and *infrastructure*. The innovative aspects of the UMM are in the structure of these parts and their inter-relations. UMM provides an opportunity to bridge gaps that currently exist in the standards arena. For example, the CORBA Component Model and J2EE component models are consistent, and yet, because of the absence of a formal meta-model, it is difficult during the evolution of each to recognize when the boundaries that maintain the consistency are crossed. Similarly, it has been demonstrated in numerous products that the COM and CORBA component models are similar (in an abstract sense) enough to allow meaningful bridging. It is, however, not possible to point to a meta-model that constrains the implementations of these technologies so that bridging is assured in practice.

Component

In UMM, components are autonomous entities, whose implementations are non-uniform, i.e., each component adheres to a distributed-component model but there is no notion of a unified implementation framework. Each component has a state, an identity, a behavior, a well-defined interface and a private implementation. In addition, each component has three aspects: 1) computational, 2) cooperative, and 3) auxiliary.

The computational aspect reflects the task(s) a component carries out. In a DCS, components must be able to 'understand' the functionality of other components. Thus, each UMM component supports the introspection, by which it precisely describes its services to others. UMM takes a mixed approach to indicate the computational aspect of a component – a simple textual part, called *inherent attributes* and a formal precise part, called *functional attributes*. The inherent attributes contain the book-keeping information about a component (e.g., author, version, etc.); while the functional part is formal and indicates precisely the computation, its associated contracts and the level(s) of service the component offers. Both the inherent and functional attributes are specified by the component's creator.

In UMM, components are always in the process of cooperating with each other. This is depicted in the cooperative aspect of each component. Informally, the cooperative aspect of a component contains: i) Pre-processing collaborators – other components on which this component depends, and ii) Post-processing collaborators – other components that may depend on this component.

In addition to computation and cooperation, mobility, security, and fault tolerance are necessary features of a DCS. The auxiliary aspect of a component addresses these features. In UMM, each component can be potentially mobile. The mobility of the component is indicated as a mobility attribute. Similarly, the security and fault-tolerant attributes of a component contain the necessary information about its security and fault-tolerance features.

Service and Service Guarantees

A service offered by a component could be an intensive computational effort or an access to underlying resources. In a DCS, it is natural to expect several choices for obtaining a specific service. Thus, each component must be able to specify the quality of the service offered.

The QoS offered by each component depends upon the computation it performs, the algorithm used, its expected computational effort, required resources, the motivation of the developer, and the dynamics of supply and demand. The QoS is an indication given by an component, on behalf of its owner, about its confidence to carry out the required services. The task of guaranteeing the necessary QoS is a key issue in any quality-oriented framework. Section 4 discusses the solutions provided by the unified approach based on UMM.

Infrastructure

Because local autonomy is inherent in a DCS, forcing every component developer to abide by certain rigid rules is doomed to fail. UMM tackles the issue of non-uniformity with the assistance of the *head-hunter* and

Internet Component Broker. These are responsible for allowing a seamless integration of different component models and sustaining cooperation among heterogeneous (adhering to different models) components.

The tasks of head-hunters are to detect the presence of new components in the search space, register their functionalities, and attempt match-making between service producers and consumers. A head-hunter is analogous to a binder or a trader in other models, with one difference – a trader is passive, while a head-hunter is active. It attempts at discovering components and registering them. During the registration process, a component informs the head-hunter about its aspects to be used during the matching process. A component may register with multiple head-hunters. Head-hunters may cooperate with each other in order to serve a large number of components.

Considering the heterogeneous nature of the components, it is conceivable that the software realization of a distributed system will require an ensemble of components adhering to different models. This requires a mediator, the *Internet Component Broker*, that will facilitate cooperation between heterogeneous components.

The Internet Component Broker (ICB) acts as a translator between two heterogeneous components. ICB utilizes adapter technology, each adapter component providing translation capabilities for specific models. Thus, an adapter component's computational aspect indicates the models for which it provides interoperability. It is expected that brokers are pervasive in an Internet environment, thus providing a seamless integration of disparate components. Adapter components register with ICB and indicate their specializations (which component models they can bridge efficiently). During a request from a seeker, the head-hunter component not only searches for a provider, but also supplies the necessary details of an ICB.

The adapter components achieve interoperability using the principles of *wrap* and *glue* technology [24]. Wrappers provide a common message-passing interface for components that frees developers from the error prone tasks of implementing interfaces and data conversions. The glue schedules time-constrained actions and carries out the actual communication between components. The automatic generation of glue and wrappers based on component specifications provides a reliable, flexible and cost-effective way to achieve interoperability.

The functionality of the ICB is analogous to that of an object request broker (ORB). The ICB provides the capability to generate the glue and wrappers necessary for components implemented in diverse component models (and providing service guarantees) to collaborate across the Internet; the ORB does this only at the level of objects written in different programming languages. An ORB defines language mappings and object adapters. An ICB provides component mappings and model adapters. While the ICB conceptually provides the capabilities of existing bridges (COM-CORBA for example), it has key features that are unique; it is designed to encompass all the aspects of components and the QoS features and associated guarantees. Thus, the ICB, in conjunction with head-hunters, provides an infrastructure necessary for scalable, reliable, and secure collaborative computation for a DCS. A preliminary version of the resource discovery service, that consists of ICBs and head-hunters, has been created and is discussed in [37, 38].

3.2.2 UA

The UA is based on the principles of UMM. The creation of a software solution for a DCS, using UA, has two levels: a) component level – developers create components, test and validate the appropriate QoS and deploy the components on the network, and b) system level – a collection of components, each with a specific functionality and QoS, and a semi-automatic generation of a software solution for the particular DCS is achieved. These two levels and associated processes are described below.

Component Development and Deployment Process

The component development and deployment process starts with a UMM requirements specification of a component from a particular domain. This specification is in a natural language and indicates the functional (i.e., computational, cooperative and auxiliary aspects) and non-functional (i.e., QoS constraints) features of the component. This specification is then refined into a formal specification. The refinement is based upon the theory of Two-Level Grammar (TLG) and natural language specifications [7, 8]. The refinement is achieved by the use of conventional natural language processing techniques (e.g. [20]) with a domain knowledge base [23] TLG specifications allow for the generation of the interface (possibly multi-level) for a component. This interface incorporates all UMM-aspects of a component. The developer then provides the implementation to all the methods indicated in the interface. This process is followed by the validation against requirements specifications. If the results are satisfactory then it is deployed on the network and is discovered by one or more head-hunters. If the component does not meet the requirement specifications then the developer refines either the UMM requirements specification or the implementation and the cycle repeats.

Formal Specification of Components in UMM

Since the UMM specifications are informally indicated in a natural language like style, UA aims at translating these into more formal specifications using TLG. TLG is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The reason that TLG is chosen is that it allows queries over the knowledge base to be expressed in a natural language like manner which is consistent with the way in which UMM is expressed. TLG is then a framework under which natural language may be used to both describe and inquire about the nature of components and systems. More details of TLG, which facilitate a formal specification of components and queries, are described in section 5.

Automated System Generation

In general, different developers will provide on the Internet a variety of possibly heterogeneous components oriented towards a specific problem domain. Once all the components necessary for implementing a specified distributed system are available and a specific problem is formulated, then the task is to assemble them into a solution. The proposed framework takes a pragmatic approach, based on Generative Programming [4, 13], to component-based programming. It is assumed that the generation environment will be built around a generative domain-specific model (GDM) supporting component-based system assembly. The distinctive features of the proposed approach are as follows:

1. The developer of the desired distributed system presents to this process a system query, in a structured form of natural language, that describes the required characteristics of the distributed system. The query is processed using the domain knowledge (such as key concepts from a domain) and a knowledge-base containing the UMM description (in the form of a TLG) of the components for that domain. From this query a set of search parameters is generated which guides head-hunter agents for a component search in the distributed environment.
2. The framework, with the help of the infrastructure, collects a set of potential components for that domain, each of which meets the QoS requirement specified by the developer. After the components are fetched, the system is assembled according to the generation rules embedded in the generative domain model. Essentially, the generated code constitutes the glue/wrapper interface between the components. The TLG formalism is used to describe the generative rules (see section 6 for further discussion) and the output of the TLG will provide the desired target code (e.g., glue and wrappers for components and necessary infrastructure for distributed run-time architecture).
3. Along with the generated system will be a formal UMM specification of the generated system so that it may be used in subsequent assemblies. This formal UMM specification will also be a basis for generating a set of test cases to determine whether or not an assembly satisfies the desired QoS.
4. The static QoS parameters are processed during generation time and hence will be processed by the TLG directly. Dynamic QoS parameters result in instrumentation of generated target code based on event grammars, which at run time will produce the corresponding QoS dynamic metrics.

To summarize, the inputs for the system assembly and generation step are: the query for the system build, UMM descriptions of the components found by headhunters, and the QoS parameters for the system build. The outputs are the generated code instrumented for the dynamic QoS metric evaluation and auxiliary code needed to compile, assemble and run the system, and UMM description of the generated system which makes it possible to add the new component to the component database. Two-Level Grammar is the formalism for representing UMM's, GDM's, QoS parameters, supporting queries, and generation rules. Only the queries that have counterparts in the GDM are processed. The GDM contains generation rules for system assembly from the components. The query language becomes an essential part of the proposed approach since the query provides the input for component search via the headhunter mechanism and following glue and wrapper generation. The query supplies the initial parameters for the headhunters to search in the distributed environment and gives the input for the generation step itself.

The proposed approach to Generative Programming besides the domain-specific generative models involves yet another dimension: components and their attributes found in the distributed environment. Since the environment is changing, the results of a query depend on the component resources available. The attributes found in the UMM descriptors of the fetched components determine the hierarchy of generation rule calls and hence the architecture of the assembled system. This implies that the UMM descriptor has to be generation-oriented, i.e. contain attributes specific for the generation needs. The generation rules represent typical design patterns for the selected domain and more general software design patterns, e.g. as advocated in [15].

QoS parameters given in the query provide yet another aspect for the generated code - the instrumentation necessary for the run-time QoS metrics evaluation. Static QoS parameters are processed at generation time by corresponding rules within the domain model. Since dynamic QoS metrics can be calculated only for particular inputs, in order to find the best possible approximation for the system, the following approach is suggested. Based on the query or informal requirements, the user has to come up with a representative set of test cases. Next the implementation is tested using the set of test cases to verify that it meets the desired QoS criteria. If it does not, it is discarded. After that, another implementation is chosen from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is found, or until the collection is exhausted. In the latter case, the process may request additional components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. If a satisfactory implementation is found, it is ready for deployment.

The same GDM is used to generate the final optimized version of the required system and UMM description of the system if the system is to be used as a stand-alone component.

4 QoS-based Approach

The UA to assuring the QoS of a DCS is made up of three steps: a) the creation of a catalog for QoS parameters (or metrics), b) a formal specification of these parameters, and c) a mechanism for ensuring these parameters, both at each individual component level and at the entire system level. In next few sections, these three steps are described in detail.

4.1 A Catalog of QoS Parameters

There are many possible QoS parameters that a component (and its developer) can use to indicate the associated service. In UA, as a first step, a catalog of QoS parameters is created [6]. The format of this catalog is based on that of the design patterns [15] catalog. This catalog provides a vocabulary for a QoS-based approach. A QoS parameter is entered into this catalog only if it is completely different from the existing ones and appears in many application domains. It is expected that this catalog will gradually evolve over a span of time.

The goal of creating the QoS catalog is two-fold: a) it assists the component developer (or the system integrator) in selecting the necessary QoS parameters for the component (or system) under construction, and b) it enables the developer (or integrator) to ensure the necessary QoS guarantees by integrating the selected QoS parameters into the assurance process.

4.1.1 Description of QoS Parameters

Each parameter is described by using the following features:

1. Name: indicates the name of the parameter.
2. Intent: indicates the purpose of the parameter.
3. Description: provides a brief informal description of the parameter.
4. Influencing Factors: depicts the factors on which the parameter depends along with their measures and degree of influence, if any.
5. Measure: indicates the unit in which to measure the parameter.
6. Known Usages: describes the known usages of the parameter.
7. Aliases: indicates other prevalent names, if any.
8. Related Parameters: indicates other related QoS parameters.
9. Consequences: indicates the effects if this parameter is used in describing the QoS of a component.
10. Levels: indicates possible QoS levels offered by a component.
11. Technologies: indicates the underlying technologies.
12. Applications: indicates the application domains in which the parameter has been used.
13. Exceptions: indicates the possible error situations and associated exception handling capabilities.
14. Example Scenario: indicates a possible scenario where it is appropriate for the parameter to be used.

4.1.2 List and Brief Description of QoS Parameters

In [42], a few QoS parameters for objects are described. That list has been augmented to create a current version of the catalog that contains the following parameters:

1. Throughput: indicates the efficiency or speed of a component (e.g., user-interaction component).
2. Capacity: indicates the maximum number of concurrent requests a component can serve (e.g., server component).
3. End-to-End Delay: indicates the time difference between the invocation of a method of a component to its completion (e.g., numerical computational component).
4. Parallelism Constraints: indicates whether a component can support synchronous or asynchronous invocations (e.g., server component).
5. Availability: indicates the duration when a component is available to offer a particular service (e.g., classifier component).
6. Ordering Constraints: indicates the order of the return results and its significance (e.g., transaction component).
7. Error Rate: indicates the probability of returning incorrect results or no result at all (e.g., arithmetic computational component).
8. Security: indicates the security-related details of a component (e.g., e-commerce component).
9. Transmission: indicates the quality of the data communication provided by a component (e.g., a routing component).
10. Adaptivity: indicates how a component can adapt to changing environment (e.g., information service provider component).

11. Evolvability: indicates how easily a component can evolve over a span of time (e.g., text-editor component).
12. Reliability: indicates reliability of the service offered by a component (e.g., real-time controller component).
13. Stability: indicates whether a component can provide a predictable quality (e.g., network controller component).
14. Result: indicates quality of the results returned (e.g., numerical computational component).
15. Achievability: indicates if a component can provide a higher degree of service than promised (e.g., multi-media transmission component).
16. Priority: indicates if a component is capable of providing prioritized service (e.g., scheduling component).
17. Compatibility: indicates if a component is environment (e.g., platform) dependent or not (e.g., applet component).
18. Presentation: indicates the presentation aspects of the result returned by a component (e.g., database component).

4.1.3 Detailed Sample Description

Although, all the above mentioned parameters have been fully described in [6], for the sake of brevity below only one parameter, *Throughput*, is described in detail.

- Name: Throughput.
- Intent: This parameter indicates the speed of efficiency of a component.
- Description: This parameter is used to specify the number of methods or requests that a component can serve per a given time unit (e.g., second) and the classification of the requested methods based on their read/write behaviors.
- Influencing Factors: This parameter depends on the following factors:
  - Algorithms used by each method and associated complexity measures (e.g., time, space) – weight of this factor is very important.
  - Available resources and their abilities and quantities – weight of this factor is very important.
  - Operating system scheduling scheme – weight of this factor is important.
- Measure: Methods_completed/Second.
- Known Usages: FTP Server, HTTP Server, Email Server, Information Classifying System, User Interaction Environment.
- Aliases: Execution Rate.
- Related Parameters: Capacity, Parallelism Constraints, and End-to-End Delay.
- Consequences: A guarantee of a higher throughput could have an adverse effect on the resources allocated to other components running on that machine thereby deteriorating their performance.
- Levels: The possible levels for throughput could be: a) low ($<$ 50 requests completed per second), b) moderate ($<$ 500 requests completed per second), and c) high ($<$ 5000 requests completed per second).
- Technologies: RPC, RMI, etc.
- Applications: Web, E-commerce, Database, Scientific Computation.

- Exceptions: a) actual throughput is less than the one promised (`LessThanPromisedException`) – this can lead to disastrous situations in critical application domains, and b) actual throughput exceeds the promised number (`MoreThanPromisedException`) – in most cases, this will not have any adverse effect, but in some it can lead to problematic situations.
- Example Scenario: In an information filtering system, a representer component provides the service of converting a textual document into its numerical equivalent form. The representer, typically, supports a function called `represent_document()`. If such a component specifies its QoS as 15 methods_completed/second, then it indicates that the representer is able to convert 15 textual documents into their numerical forms in one second. A representer can also specify that it provides either one level (say 15 methods/second) or two levels (15 methods/sec and 30 methods/sec) or three levels (15 methods/sec, 30 methods/sec and 45 methods/sec) of services.

4.2 Empirical Analysis of QoS Parameters

In [39] an empirical study is provided that illustrates the rules for composing and decomposing QoS parameters. Here a brief discussion is provided in the context of the end-to-delay (also called as turn-around time) for a simple account system (similar to the case study presented in the section 6).

As stated earlier, the end-to-end delay is the response time of any system. Many IT professionals use the *eight-second rule* as a threshold for the maximum allowable limit for the end-to-end delay with a download. It is ranked as the second most important QoS parameter after the availability[4] and is critical for many different application domains. Many different factors affect end-to-end delay. These include, the actual computational efforts and policies, network delays, and possible security delays. End-to-end delay is a dynamic parameter and is easily composable, i.e., the total end-to-end delay for a system is a summation of the end-to-end delays of the individual components that make up the system. Thus, given the individual end-to-end delays, it is possible to predict the delay of the assembled system.

As a simple case study, to validate this model, a simple bank system consisting of three components (a client, a server and a database) was created. Each component contained end-to-end delay as one of its QoS parameters. These components were deployed and executed, in isolation, on a LAN of Sun SPARC machines and average values for their end-to-end delays were computed. These values were found to be 34 ms (for the client component), 119 ms (for the server component) and 126 ms (for the database component). Based on these numbers, it was predicted that the integrated system will have the end-to-end delay in the range of 278 ms (which is the summation of the individual end-to-end delays). A distributed system made of these three components was created, deployed and its average value for the end-to-end delay was computed. This value was found to be 287 ms – which is in the same order as the predicted value (278 ms).

This simple experiment was carried to indicate that an empirical evaluation, based on the QoS catalog and composition/decomposition rules, will enable a system integrator to not only assemble heterogeneous components but also deploy, execute and validate the combined ensemble.

Instead of a simple empirical evaluation, indicated above, the UniFrame approach will use the notion of event grammars (as described in the next section) for measuring and validating QoS parameters of an integrated system.

4.3 QoS Metrics and Implementation

Dynamic QoS metrics can be expressed in a uniform manner based on the system behavior models. In [2, 3] the use of event grammars as a basis for such models is suggested. An event is an abstraction of any detectable action performed during run-time, for instance, execute a statement or call a procedure. An event has beginning, end, and duration, and some other attributes, such as program states at the beginning and end of the event, source code associated with the event, and so on. Two binary relations are defined

[4]Hence, these two parameters are also used in the case study of section 6.

for the events. One event may precede another event, e.g. one statement execution may precede another, or one event may be included in another, e.g., a statement execution event may appear inside a procedure call event. System execution may be represented as a set of events with the two basic relations between them - event trace. An event grammar is a set of axioms that determines possible configurations of events of different types within the event trace. For example, the axiom

```
    execute-assignment : evaluate-expression  perform-destination
```

specifies that the event of the type execute-assignment contains a sequence (with respect to the precedence relation) of events of types evaluate-expression and perform-destination, correspondingly.

Different dynamic QoS metrics could be expressed as appropriate computations over event traces. For example, if 'function-call IS A' denotes an event of the type function call with the name A, then the total duration of this function call may be expressed as:

```
   SUM/[ X: function-call IS A FROM execute-program Duration(X)]
```

`[...]` denotes a sequence constructor which selects from the whole event trace (an event of the type execute-program) all events that match the pattern function-call IS A, takes the Duration attribute of those events, and sums them up. Event grammars and the notion of the computations over event traces provide a uniform framework to define different dynamic QoS metrics. This mechanism may be a basis of automatic instrumentation of the generated code.

As has been indicated above, static QoS parameters are processed by generation rules at generation time according to the inference rules encoded in the domain model (see example in section 6.5). Therefore, the event grammar and a language for event trace computations are part of the GDM.

It should be noted that the assurance of QoS (as described above) indicates that a component can guarantee appropriate values for its QoS parameters in an 'ideal' situation. This does not guarantee that a component will be able to either provide this QoS under failure circumstances or will automatically adjust its QoS to hide the failures. For the failure situations, the ideas provided by QuO [5] or RAPIDware [27] can be incorporated into the UMM and UA.

5 Formal Specification in the Unified Approach

Formal specification in UA is by means of Two-Level Grammar (TLG, also called W-grammar). TLG was originally developed as a specification language for programming language syntax and semantics and was used to completely specify ALGOL 68 [41]. TLG may be used as an object-oriented requirements specification language and also serve as the basis for conversion from requirements expressed in natural language into a formal specification [22]. This section describes the TLG language details that facilitate these processes and elaborates on how the language may be used in formal specification of UMM specifications.

The name "two-level" in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars interacting in a manner such that their combined computing power is equivalent to that of a Turing machine [36]. These two grammars define the set of type domains and the set of function definitions operating on those domains. Note that while the term "domain" is used in a type-theoretic context, the notion can be scaled up to a much larger context as in domain of "objects." These grammars may be defined in the context of a class in which case the type domains define the instance variables of the class and the function definitions define the methods of the class. Each of these terms are defined below.

5.1 Types

The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. The function domains of TLG may be formally structured as linear data structures such as lists, sets, bags, or singleton data objects, or be configured as tree-structured data objects. The standard structured data types of product domain and sum domain may be treated as special cases of these.

Domain declarations have the following form:

Identifier-1, Identifier-2, ..., Identifier-m ::
`data-object-1; data-object-2; ...; data-object-n.`

where each `data-object-i` is a combination of domain identifiers, singleton data objects, and lists of data objects, which taken together as a union form the type of *Identifier-1*, *Identifier-2*, ..., *Identifier-m*. Note that if n=1, then the domain is a true singleton data object, whereas if n>1, then the domain is a set of the n objects. Syntactically, domain identifiers are capitalized, with underscores or additional capitalizations of successive words for readability (e.g., *IntegerList*, *Symbol_Table*, etc.), and singleton data objects are finite lists of natural language words written entirely in lower case letters (e.g., `sorted list`). For improved readability, the domain identifiers are represented in italics and data objects are represented in the typewriter font.

A list, set or bag structure is denoted by a regular expression or by following a domain identifier with the suffix *List*, *Set*, or *Bag*, respectively. Following conventional regular set notation, * implies a set of zero or more elements while + denotes a set of one or more elements. As in any programming language, readability is promoted through the use of appropriate names for identifiers. Furthermore, there exists a predefined environment of primitive types, defining such domains as *Integer*, *Boolean*, *Character*, *String*, etc., in the obvious ways. The main difference between list structures and tree structured domains in terms of their declaration is whether the defining domain identifier declaration is recursive or not. Recursive domains are more powerful in that they allow "context-free" data types to be defined, such as expression strings with balanced parentheses.

5.2 Functions

Function definitions comprise the operational part of a TLG specification. Their syntax allows for the semantics of the function to be expressed using a structured form of natural language. Function definitions take the forms:

function signature.
function signature : function-call-1, function-call-2, ..., function-call-n.

where n≥1. Function signatures are a combination of natural language words and domain identifiers. For improved readability, we will use boldface type to represent the function keywords. Domain identifiers in the context of a function typically correspond to variables in a conventional logic program. As in logic programs, some of these variables will typically be input variables and some will be output variables, whose values are instantiated at the conclusion of the function call. Therefore, functions usually return values through the output variables rather than directly, in which case the direct return value is considered as a Boolean `true` or `false`. `true` means that control may pass to the next function call, while `false` means the rule has failed and an alternative rule should be tried if possible. Alternative rules have the same format as that given above. If multiple function rules have the same signature, then the multiple left hand sides may be combined with a ; separator, as in:

function signature :
function-call-11, function-call-12, ..., function-call-1j;
function-call-21, function-call-22, ..., function-call-2k;
...
function-call-n1, function-call-n2, ..., function-call-nm.

where there are n alternatives, each having a varying number of function calls. Besides Boolean values, functions may return regular values, usually the result of arithmetic calculations. In this case, only the last function call in a series should return such a value, i. e., not **function-call-1**, **function-call-2**, ..., **function-call-(n-1)**.

An important aspect about TLG is that the functions may be written at a very high level of abstraction (e.g. **compute the total mass and total cost**) or embedded into a domain definition as in traditional object-oriented programs (e.g. **compute the** *TotalMass* **and** *TotalCost* of *This Part* **by computing the** *TotalMass* **and** *TotalCost* **of its** *Subparts*, which might be embedded as a method in a *Part* class). The use of natural language in the function may be regarded as a form of infix notation for functions, in contrast with the customary prefix forms of most other programming languages. It is similar to multi-argument message selectors in Smalltalk but provides even greater flexibility, including the presence of logical variables, denoted by the use of domain names (capitalized). This notation provides a highly readable way of writing what is to be done and is wide-spectrum in the sense that "what is to be done" may be expressed at multiple levels. The functions typically return a Boolean value as the main operation is to instantiate the logical variables as in Prolog, but simple function values such as arithmetic expressions may also be computed.

To explain the operational semantics of Two-Level Grammar function rules, note that each function call on the right hand side of a function definition should correspond to a function signature defined within the scope of the TLG program or be a special operation such as a Boolean comparison, assignment statement, or if-then-else statement. The most important aspect of function definitions is that every domain identifier with the same name is instantiated to the same value, as in Prolog. This is called *consistent substitution.* If variables have the same root name but are numbered, then the numbers are used to distinguish between variables. A numbered variable *V1* will then be different from a variable *V2* and the two can have different values. However, they will be of the same type, namely type *V*. Note that once a variable has been assigned a value, it may not be reassigned, unless it is an instance variable of a class, and even in this case, it would not be usual to do so in the same function. Each function definition may therefore be thought of as a set of logical rules. Also, as in Prolog, the function calls are executed in the order given in the function definition. Functions may be recursive with the expected operational behavior.

Besides defined functions, TLG supports the usual arithmetic and Boolean operations. For lists, list comprehensions are also supported as are iterators over the list. The syntax of a list comprehension is **list all** *Element* **from** *ElementList1* **such that** *Element* **condition giving** *ElementList2*. This returns a list, *ElementList2*, of all *Element* values in *ElementList* satisfying the given condition. The syntax of an iterator is **select** *Element* **from** *ElementList* **with** *Element* **condition**. This returns the first *Element* from *ElementList* which satisfies the condition.

5.3 Classes

In order to support object-orientation, TLG domain declarations and associated functions may be structured into a class hierarchy supporting multiple inheritance. The syntax of TLG class definitions is:

class *Identifier-1* [**extends** *Identifier-2*, *Identifier-3*, ... *Identifier-n*].
    instance variable and method declarations
end class [*Identifier-1*].

In the above syntax, square brackets are used to indicate the construct is optional. *Identifier-1* is declared to be a class which inherits from classes *Identifier-2*, *Identifier-3*, ..., *Identifier-n*. Note that the **extends** clause is optional so a class need not inherit from any other class. The instance variables comprising the class definition are declared using the domain declarations described earlier. In general, the scope of these domain declarations is limited to the class in which they are defined, while the methods, corresponding to TLG function definitions, have scope anywhere an object of the given class is referred to. These notions of scoping correspond to *private* and *public* access respectively in object-oriented languages such as C++ and

Java, and either scope may be declared explicitly or the scope may be made *protected*. Methods are called by writing a sentence or phrase containing the object. The result of the method call is to instantiate the logical variables occurring in the method definition.

In any class for every instance variable of simple type there are **get** and **set** methods to access or modify that variable.

TLG class declarations serve to encapsulate the TLG domain declarations and function definitions. The class hierarchy which is resident in TLG is a small forest of built-in classes, such as integers, lists, etc. The "main" program is nothing more than a set of object declarations using the existing class identifiers as domain names and a "query" of the appropriate methods.

5.4 Example

As an example of a TLG specification, consider the following translation scheme for producing three address code [1] from simple arithmetic expressions.

```
class CodeGenerator.
    Expression :: Term {AddingOperator Term}*.
    AddingOperator :: +; -.
    Term :: Factor {MultiplyingOperator Factor}*.
    MultiplyingOperator :: *; /.
    Factor :: ( Expression ); Identifier; Float; Integer.
    ExpressionIdentifier, TermIdentifier, FactorIdentifier, Identifier :: String.
    ExpressionType, TermType, FactorType, Type :: float; integer; undefined.

    three address code for Expression AddingOperator Term is Identifier type Type :
        three address code for Expression is ExpressionIdentifier type ExpressionType,
        three address code for Term is TermIdentifier type TermType,
        common type of ExpressionType and TermType is Type,
        type convert ExpressionIdentifier type ExpressionType into Identifier1 type Type,
        type convert TermIdentifier type TermType into Identifier2 type Type,
        ThreeAddressCode generate temporary Identifier := Identifier1 AddingOperator Identifier2.

    three address code for Term MultiplyingOperator Factor is Identifier type Type :
        ... similar to above ...

    three address code for ( Expression ) is Identifier type Type :
        three address code for Expression is Identifier type Type.

    three address code for Identifier is Identifier type Type :
        SymbolTable lookup Identifier giving Type,
        Type != undefined.

    three address code for Float is Float type float.

    three address code for Integer is Integer type integer.
```

....

end class.

For simplicity only two types, **float** and **integer**, are assumed. There is also a *SymbolTable* class assumed with standard operations such as looking up an identifier to obtain its type, and a *ThreeAddressCode* class assumed with an operation to generate a three-address code instruction in the code array, possibly including an assignment to a temporary variable. Rules to check type compatibility and perform type conversions are also present but not shown here. Error checking is not explicitly indicated but would occur through failure of any rule, e.g., a syntactically ill-formed expression would not match any of the **three address code** rules, an identifier not declared would cause the identifier rule to fail, and any errors in typing would cause the type checking rules to fail.

These rules would be queried as follows:

CodeGenerator **three address code for** `a * (b + 1)` **is** *Id* **type** *Type*

This creates a code string of:

```
t1 := b + 1
t2 := a * t1
```

and returns `t2` for *Id* and `integer` for *Type*, respectively (assuming that `a` and `b` are stored in the symbol table as type `integer` variables).

This example illustrates that TLG may be used to provide for attribute evaluation and transformation, syntax and semantics processing of languages, parsing, and code generation. All of these are required to use TLG as a specification language for generative rules [9].

5.5 Implementation

Two-Level Grammar is implemented as part of a specification development environment which facilitates the construction of TLG specifications from natural language, and then translates TLG specifications into executable code. The natural language requirements are translated into TLG through Contextual Natural Language Processing (CNLP) [26] which constructs a knowledge representation of the requirements which may then be expressed using TLG [23]. The TLG is then translated into VDM++ [17], the object-oriented extension of the Vienna Development Method (VDM) specification language [21]. The IFAD VDM Toolbox^{TM} [18] may then be used to generate code in an object-oriented programming language such as Java or C++.

6 A Case Study

This section presents a simple example from the account management domain to illustrate how the previous concepts can be applied to assemble a component-based system from a developer's request. Before the developer can make this request, experts must construct a GDM for the domain of interest and suppliers must provide on the network any UMM components that might be necessary to meet the developer's needs.

6.1 TLG Component Specification

Two-Level Grammar is used as the formalism for both the UMM and the generative rules, which make up part of the GDM. The UMM formalization establishes the context for which the generative rules may be applied.

6.1.1 UMM

The basic TLG statement of the UMM specification template for components in the example is given below. Some details are omitted for brevity. The domain experts create the UMM specification which may then be parsed into TLG according to the template below. Any component described using UMM will be typed according to these declarations. A supplier implements a component and then makes it available to potential users by publishing a description of it that this TLG can parse.

UMM :: *ComponentName InformalDescription FunctionList ComputationalAttributes*
  CooperationAttributes AuxiliaryAttributes QoSMetricList.
ComponentName, *InformalDescription*, *Function* :: *String.*
ComputationalAttributes :: *InherentAttributes FunctionalAttributes.*
InherentAttributes :: *Id Version DateDeployed.*
Id :: *String.*
Version :: *Float.*
DateDeployed :: *Date.*
FunctionalAttributes :: *TaskDescription AlgorithmAndComplexity SyntacticConstruct Technology.*
TaskDescription :: *String.*
AlgorithmAndComplexity :: ....
SyntacticConstruct :: *FunctionSignatureList.*
FunctionSignature :: ....
Technology :: `corba`; `java applet`; `java rmi`; ....
CooperationAttributes :: *PreprocessingCollaboratorList PostprocessingCollaboratorList.*
PreprocessingCollaborator :: *String.*
PostprocessingCollaborator :: *String.*
AuxiliaryAttributes :: ....
QoSMetric :: *Throughput*; *Capacity*; *EndToEndDelay*; *ParallelismConstraints*; *Availability*; ....
Throughput :: *Float.*
Capacity :: *Integer.*
EndToEndDelay :: *Integer* `ms`.
ParallelismConstraints :: `synchronous`; `asynchronous`.
Availability :: *Float*; *Integer* `%`.
....

6.2 Client and Server Distributed Components

At this point in the example, suppose that suppliers have implemented and made available within UMM three types of components that developers can use to assemble account management systems with a client/server architecture. These include two instances of `AccountServer` and one instance of `AccountClient`. The server components are heterogeneous – `javaAccountServer` adheres to the Java-RMI model; while `corbaAccountServer` is developed using the CORBA model. The client, `javaAccountClient` is developed by using the Java-RMI model and is implemented as an applet. The partial UMM descriptions of these components are presented below. One can see that the previous TLG component specification can parse these declarations.

6.3 Component Descriptions in UMM

```
javaAccountServer
-----------------
Informal Description: Provides an account management service. Supports three
functions: javaDeposit(), javaWithdraw() and javaBalance().

1. Computational Attributes:
    a) Inherent Attributes:
        a.1 id: intrepid.cs.iupui.edu/jServer

    b) Functional Attributes:
        b.1 Acts as an account server
        b.2 Algorithm: simple addition/subtraction
        b.3 Complexity: O(1)
        b.4 Syntactic Contract:
            void javaDeposit(float ip);
            void javaWithdraw(float ip) throws overDrawException;
            float javaBalance();
        b.5 Technology: Java-RMI
              .......

2. Cooperation Attributes:
    2.1) Pre-processing Collaborators: AccountClient

3. Auxiliary Attributes:
              .......
4. QoS Metrics:
    Availability: 90%
    End-to-End Delay < 10 ms

corbaAccountServer
------------------
Informal Description: Provides an account management service. Supports three
functions: corbaDeposit(), corbaWithdraw() and corbaBalance().

1. Computational Attributes:
    a) Inherent Attributes:
        a.1 id: jovis.cs.iupui.edu/coServer

    b) Functional Attributes:
        b.1 Acts as an account server
        b.2 Algorithm: simple addition/subtraction
        b.3 Complexity: O(1)
        b.4 Syntactic Contract:
            void corbaDeposit(float ip);
            void corbaWithdraw(float ip) throws overDrawException;
            float corbaBalance();
        b.5 Technology: Java-CORBA
              .......
```

```
2. Cooperation Attributes:
    2.1) Pre-processing Collaborators: AccountClient

3. Auxiliary Attributes:
             .......

4. QoS Metrics:
    Availability: 95%
    End-to-End Delay < 10 ms

javaAccountClient
--------------
Informal Description: Requests account services from an appropriate server and
interacts with the user -- implemented as a web-based applet. Supports
functions: depositMoney(), withdrawMoney() and checkBalance().

1. Computational Attributes:
    a) Inherent Attributes:
        a.1 id: galileo.cs.iupui.edu/aClient

    b) Functional Attributes:
        b.1 accepts user queries and presents the results using a GUI
        b.2 Algorithm: Java Foundation Classes (JFC)
        b.3 Complexity: O(1)
        b.4 Syntactic Contract
            void depositMoney(float ip);
            void withdrawMoney(float ip);
            float checkBalance();
        b.5 Technology: Java Applet
        .......

2. Cooperation Attributes:
    2.1) Post-processing Collaborators: AccountServer

3. Auxiliary Attributes:
             .......

4. QoS Metrics:
    Availability: 80%
    End-to-End Delay < 20 ms
```

6.4 Account Management Problem Statement

Once a GDM for an accounting management domain has been defined and appropriate UMM components are available, a developer can pose the problem of finding and assembling the components necessary to create an account management system. Resolving this query involves identifying in the GDM a design for such a system (this part of the GDM is not shown here), issuing a request to the UMM headhunters for component implementations that match the UMM component specifications of the design, and assembling them into an implementation of a problem solution. An important part of the query statement is the identification of the application domain in which the solution design lies.

A sample query for the present example can be informally stated as: *Create an account management system that has: availability >= 50% and end-to-end delay < 50 ms.* This query specifies that a static and a dynamic QoS parameter must be satisfied. The natural language processor of UA will infer that the application domain is account management and, thus, conforms to the example GDM described above. In response, the UMM headhunters will discover components for the following system assemblies:

1. Java-Java System
   (a) `javaAccountClient` – availability >= 80%, End-to-End delay < 20ms, Java Applet Technology
   (b) `javaAccountServer` – availability >= 90%, End-to-End delay < 10ms, Java-RMI technology
   (c) Infrastructure Needed – JVM and Appletviewer
2. Java-CORBA System
   (a) `javaAccountClient` – availability >= 80%, End-to-End delay < 20ms, Java Applet Technology
   (b) `corbaAccountServer` – availability >= 95%, End-to-End delay < 10ms, Java-RMI technology
   (c) Infrastructure Needed – JVM, Appletviewer, ORB, Java-CORBA bridge

6.4.1 Generation Rules

The process of parsing UMM component descriptions, as mentioned in the preceding examples, provides a structure to the UMM that can be processed by TLG functions. These functions include generative rules for construction of the wrapper/glue code and the event grammar instrumentation to assure the QoS of the accounting system.

A sampling of TLG rules that may be used to generate the appropriate glue/wrapper code to connect the components of the accounting system are presented below. These rules are based on selecting from the GDM of the accounting systems the appropriate system model for this two-component DCS.

ClientUMM, ServerUMM :: *UMM.*
ClientOperations, ServerOperations :: *{Interface}*.*
generate system from *ClientUMM* **and** *ServerUMM* :
  ClientOperations := *ClientUMM* **get operations,**
  ServerOperations := *ServerUMM* **get operations,**
  OperationMapping := **map** *ClientOperations* **into** *ServerOperations,*
  ComponentModel := *ServerUMM* **get component model,**
  generate java code for *OperationMapping* **using** *ComponentModel.*

This rule generates Java code for two UMM models representing a client and server, respectively. For this example, the *ClientUMM* would be the UMM specification of `javaAccountClient` presented previously and the *ServerUMM* would be the UMM specification of `javaAccountServer` or `corbaAccountServer`. The main tasks are to map client operations onto server operations, e.g., `depositMoney` in `javaAccountClient` maps to `corbaDeposit` in `corbaAccountServer` or to `javaDeposit` in `javaAccountServer`, and then generate the code to implement this mapping. The generated code will be in Java since the client code is in Java and must seamlessly interface with it. If the client is in C++ or other language, similar rules will be defined and many rules will be language independent.

The actual mapping to be defined will be based upon a natural language analysis of the names of operations. The closer the names match, the more easily the system can establish the correct mapping. This depends upon both the care and style with which the user has written the interface method names and so may vary widely. For this example, it can be seen that the correspondence between names, while not exact, is relatively close.

The next rule describes the specifics of generating CORBA code in Java to implement the mapping that arises by combining the `javaAccountClient` with the `corbaAccountServer`.

```
CorbaPackageName, CorbaObjectType, CorbaObjectName :: String.
ClassName, JavaClassName :: String.
generate java code OperationMapping using corba :
  CorbaPackageName := OperationMapping get corba package name,
  CorbaObjectClass := OperationMapping get corba object type,
  ClassName := OperationMapping get class name,
  JavaClassName := Java || ClassName,
  CorbaObjectName := object || ClassName,
  SetUpCode := ComponentModel generate java code,
  Operations := generate java code for OperationMapping,
  return
    import CorbaPackageName .  *;
    public class JavaClassName {
      private CorbaObjectClass CorbaObjectName ;
      // initialize CORBA client module
      public void init () {
        SetUpCode
      }
      Operations
    }.
```

This rule generates the class structure required by the Java implementation, which consists of a function `init` to set up the CORBA ORB and the operations needed in the server. This includes the code to initialize the CORBA object so that future operations can refer to it. It is necessary to first extract the names of the CORBA package, class of the CORBA object to be referenced within the package, and the name of the class itself. These are all stored in the *OperationMapping*. The name of the Java class generated is simply the string "Java" concatenated [5] with the name of the server class, i.e., *JavaCorbaAccountServer*. The name of the CORBA object is generated in a similar way.

The rule below describes the mechanism for generating the individual methods in *JavaCorbaAccountServer*. For simplicity, only the case where the class is to contain a single method is shown. Multiple methods would be handled in a similar manner.

```
generate java code for OperationName1 ArgumentList1 ReturnType
    maps to OperationName2 ArgumentList2 ReturnType :
  JavaReturnType := java type of ReturnType,
  JavaArgumentList :=
    list all Argument from ArgumentList1
      mapped by function java argument of Argument,
  JavaArgumentListDefinition := separate JavaArgumentList by , ,
  OperationCall := generate java code for OperationName2 ArgumentList1 ReturnType,
  return
```

[5]The TLG concatenation operation (||) differs from juxtaposition in that it does not produce a space between the operands.

```
public JavaReturnType OperationName1 ( JavaArgumentListDefinition ) {
    EventTrace .  setBeginTime ();
    OperationCall
    EventTrace .  setEndTime ();
    EventTrace .  calculateResponseTime ();
}.
```

This generation assumes that the methods have the same return type and so the main task is to express the arguments of the first operation in terms of Java syntax and generate the appropriate method call. The former is accomplished by using a TLG list comprehension to map the arguments in *ArgumentList1* into corresponding Java arguments represented by *JavaArgumentList*. There is a subtlety here in that *JavaArgumentList* is an abstract syntax representation of the desired argument list and so this must be made into concrete syntax using the **separate** operation which adds the appropriate commas in between the argument declarations. The appropriate method call is handled by the rule below.

```
generate java code for OperationName ArgumentList ReturnType :
    IdentifierList :=
        list all Argument from ArgumentList
            mapped by function argument id of Argument,
    IdentifierListInCall := separate IdentifierList by , ,
    return
        CorbaObjectName . OperationName ( IdentifierListInCall );.
```

Again a list comprehension is used to extract the arguments from the argument list, this time only the identifier part (achieved by **function argument id of** *Argument*). Likewise, the abstract syntax representation must be made concrete by comma separators.

Finally, the event grammar instrumentation is added to measure the time at the beginning of the server method call and again at the end so that the actual response time can be evaluated against the required QoS ($< 100ms$). The QoS metrics for "response delay" mean execution time for each method call within the server or client, and require the instrumentation of each generated wrapper for the client/server method call with auxiliary functions able to check the clock at the beginning and at the end of method call, calculate the duration, and submit it to the execution monitor (also generated as a part of instrumentation). It is assumed that these are taken care of by a class called `EventTrace`. Each of the two example systems will be implemented with the code for carrying out event trace computations according to test cases which must be supplied by the user. These test cases will be executed to verify that the bank account management system satisfies the QoS specified in the query. If the system is not verified, it is discarded. This verification process is carried out for each of the generated bank account management system (two in the above example). Then the one with the best QoS is chosen, in the above example the `CorbaAccountServer` and `JavaAccountClient` combination.

For the example UMM specification, the following code for the `depositMoney` function would be produced.

```
public void depositMoney (float ip) {
  EventTrace . setBeginTime ();
  objectCorbaAccountServer . deposit (ip);
  EventTrace . setEndTime ();
  EventTrace . calculateResponseTime ();
}
```

6.5 QoS

Each component has two QoS parameters - 1) static - run-time availability (e.g. 90% and 95% respectively) and 2) dynamic - end-to-end delay measured in milliseconds. The desired QoS of the assembled system includes both of these parameters as well. For this reason the GDM will contain a rule for the static parameter that will multiply the various availability parameters (e.g. obtaining 85.5% availability for the assembled system in this case), assuming component availability is independent.

For the dynamic parameter, the generator will provide the necessary instrumentation for taking the clock and calculating the end-to-end delay at run-time. The knowledge about metrics for the QoS parameter 'end-to-end delay' is represented in terms of the `Duration` attribute for events of the type method-call, and the generic computation over the event trace that takes the clock and sums up those durations yielding a measured end-to-end delay time for the accounting system.

One of the two example systems, mentioned in the section 6.4, will be implemented with the code for carrying out event trace computations according to user supplied test cases. These test cases will be executed to verify that the accounting system satisfies the QoS specified in the query of the section 6.4. If the system is not verified, it is discarded. This verification process is carried out for each of the generated accounting systems (two in the above example). Then one with satisfactory QoS is chosen; in the above example this is the `corbaAccountServer` and `javaAccountClient` combination.

7 Conclusion

This paper has presented a framework that allows an interoperation of heterogeneous and distributed software components. This framework incorporates the following key concepts: a) a meta-component model, b) integration of QoS at the individual component and distributed system levels, c) validation and assurance of QoS, based on the concept of event grammars, d) formal specification, based on Two-Level Grammar, of each component and associated queries for integrating a distributed system, and e) generative rules, along with their formal specifications, for assembling an ensemble of components out of available choices. The software solutions for future DCS will require either automatic or semi-automatic integration of software components, while abiding by the QoS constraints advertised by each component and the system of components. The result of using UMM and the associated tools is a semi-automatic construction of a distributed system. Although a simple case study is provided in this paper, the principles of the proposed approach are general enough to be applied to larger cases. Experimentation with such examples is necessary to establish the extent to which such scale up is feasible in practice.

Acknowledgments. The material presented in this paper is based upon work supported by, or in part by, a) the U. S. Office of Naval Research under award number N00014-01-1-0746, b) the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number 40473-MA, and c) the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350.

References

[1] Aho, A. V. and Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Auguston, M. Program Behavior Model Based on Event Grammar and its Application for Debugging Automation. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, 1995.

[3] Auguston, M. and Gates, A. and Lujan, M. Defining a Program Behavior Model for Dynamic Analyzers. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, pages 257–262, 1997.

[4] Batory, D., Chen, G. and Robertson, E., and Wang, T. Design Wizards and Visual Programming Environments for GenVoca Generators. *IEEE Transactions on Software Engineering*, pages 441–452, 2000.

[5] BBN Corporation. *Quality Objects (Quo) URL:-http://www.dist-systems.bbn.com/tech/QuO/*, 2001.

[6] Brahnmath, G. J., Raje, R. R., Olson, A. M., Auguston, M., Bryant, B. R., and Burt, C. C.,. A Quality of Service Catalog for Software Components. In *Proceedings of the Southeastern Software Engineering Conference (in press)*, 2002.

[7] Barrett R. Bryant. Object-oriented natural language requirements specification. In *Proceedings of ACSC 2000, the 23rd Australasian Computer Science Conference*, pages 24–30, 2000.

[8] Bryant, B. R. and Lee, B.-S. Two-Level Grammar as an Object-Oriented Requirements Specification Language. In *Proceedings of HICSS-35, 35th Hawaii International Conference on System Sciences*, `http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf`, 2002.

[9] Bryant, B. R., Auguston, M., Raje, R. R., Burt, C. C., and Olson, A. M. Formal Specification of Generative Component Assembly Using Two-Level Grammar. In *Proceedings of SEKE 2002, Fourteenth International Conference on Software Engineering and Knowledge Engineering (in press)*, 2002.

[10] Burt, C. C., Bryant, B. R., Raje, R. R., Olson, A. M., and Auguston, M. Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models. In *Proceedings of EDOC 2002, the 6th IEEE International Enterprise Distributed Object Computing Conference (in press)*, 2002.

[11] Burt, C. C., Raje, R. R., Auguston, M., Bryant, B. R., and Olson, A. M. Quality of Service (QoS) Standards for Model Driven Architecture. In *Proceedings of the Southeastern Software Engineering Conference (in press)*, 2002.

[12] California Institute of Technology. *Caltech Infospheres On-line Documentation, URL:- http://www.infospheres.caltech.edu/*, 1998.

[13] Czarnecki, K. and Eisenecker, U. W. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[14] Fox, G. The Document Object Model - Universal Access - Other Objects - CORBA, XML, Jini, JavaScript, etc. *http://www.npac.syr.edu/users/gcf/msrcobjectsapril99*, 1999.

[15] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publication Company, 1995.

[16] Globus Project. *Globus Website, URL:- http://www.globus.org/*, 2000.

[17] IFAD. The IFAD VDM++ Language. Technical report, IFAD, 1999.

[18] IFAD. The VDM++ Toolbox User Manual. Technical report, IFAD, 2000.

[19] Israel, B. and Kaiser, G. Coordinating Distributed Components Over the Internet. *IEEE Internet Computing*, pages 83–86, 2(2), 1998.

[20] Jurafsky, D. and Martin, J. H. *Speech and Language Processing.* Prentice Hall, 2000.

[21] Larsen, P. G., et al. Information Technology - Programming Languages, Their Environments and System Software Interfaces - Vienna Development Method - Specification Language - Part I: Base Language. Report, International Standard ISO/IEC 13817-1, December 1996.

[22] Lee, B.-S., and Bryant, B. R. Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 932–936, 2002.

[23] Lee, B.-S., and Bryant, B. R. Contextual Knowledge Representation for Requirements Documents in Natural Language. In *Proceedings of FLAIRS 2002, the 15th International Florida AI Research Symposium (In Press)*, 2002.

[24] Luqi, Berzins, V., Ge, J., Shing, M., Auguston, M., Bryant, B. R. and Kin, B. K. DCAPS - Architecture for Distributed Computer Aided Prototyping System. In *Proceedings of RSP 2001, the 12th International Workshop on Rapid System Prototyping*, 2001.

[25] Manola, F. Technologies for a Web Object Model. *IEEE Internet Computing*, 3(1):38–47, January-February 1999.

[26] McCarthy, J. Notes on Formalizing Context. Technical report, Computer Science Department, Stanford University, Stanford, CA, 1993.

[27] Michigan State University. *RAPIDware: Component-Based Development of Adaptable and Dependable Middleware URL:-http://www.cse.msu.edu/rapidware/*, 2001.

[28] Microsoft. .NET home Page. *URL:- http://www.microsoft.com/net/*, 2002.

[29] Microsoft Corporation. *DCOM Specifications, URL:- http://www.microsoft.com/oledev/olecom*, 1998.

[30] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999.

[31] Object Management Group. Model Driven Architecture: A Technical Perspective. Technical report, Object Management Group Document No. ab/2001-02-01/04, February 2001.

[32] Orfali R, and Harkey, D. *Client/Server Programming with JAVA and CORBA.* John Wiley & Sons, Inc., 1997.

[33] Raje, R. UMM: Unified Meta-object Model for Open Distributed Systems. In *Proceedings of the Fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP 2000)*, 2000.

[34] Raje, R., Auguston, M., Bryant, B., Olson, A. and Burt, C. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. In *Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration*, pages 109–119, 2001.

[35] Rogerson, D. *Inside COM.* Microsoft Press, 1996.

[36] Sintzoff, M. Existence of van Wijngaarden's Syntax for Every Recursively Enumerable Set. *Ann. Soc. Sci. Bruxelles*, 2:115–118, 1967.

[37] Siram, N. An Architecture for the UniFrame Resource Discovery Service. Master's thesis, Indiana University Purdue University Indianapolis, 2002. Department of Computer and Information Science.

[38] Siram, N. N., Raje, R. R., Bryant, B. R., Olson, A. M., Auguston, M., and Burt, C. C. An Architecture for the UniFrame Resource Discovery Service. In *Proceedings of SEM 2002, the 3rd International Workshop on Software Engineering and Middleware (in press)*, 2002.

[39] Sun, C., Raje, R. R., Olson, A. M., Bryant, B. R., Auguston, M., Burt, C. C., and Huang, Z. Composition and Decomposition of QoS Parameters in Distributed Component-based Systems. In *Proceedings of the 5th IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP 2002) (in press)*, 2002.

[40] University of Virginia. *Legion Project, URL:- http://www.cs.virginia.edu/ legion*, 1999.

[41] van Wijngaarden, A. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica*, 5:1–236, 1974.

[42] Zinky, J. A., Bakken, D. E., and Schantz, R. Overview of Quality of Service for Distributed Objects. In *Proceedings of the Fifth IEEE Dual Use Conference*, 1995.

A Quality of Service Catalog for Software Components

Girish J. Brahnmath[1] Rajeev R. Raje[1] Andrew M. Olson[1] Mikhail Auguston[2] Barrett R. Bryant[3] Carol C. Burt[3]

Abstract

Component-based Software Development is being recognized as the direction in which the software industry is headed. With the proliferation of Commercial Off The Shelf (COTS) Components, this trend will continue to emerge as a preferred technique for developing distributed software systems encompassing heterogeneous components. In order for this approach to result in software systems with a predictable quality, the COTS components utilized should in turn offer a guaranteed level of quality. This calls for an objective paradigm for quantifying the quality of service of COTS components. A Quality of Service (QoS) catalog, proposed here, for software components is a first step in quantifying the quality attributes. This catalog is a critical component of the UniFrame project, which targets at unifying the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques.

Keywords: Quality of Service, non-functional attributes, QoS catalog, Component-based development.

1. Introduction:

Component-based software development uses appropriate off the shelf software components to create software systems. The notion of assembling complete systems out of prefabricated parts is prevalent in many branches of science and engineering such as manufacturing. This leads to the creation of prompt and economical products. This is possible because of the existence of standardized components that meet a manufacturer's functional and non-functional (quality) requirements. Also, the task of the manufacturer is made much easier because of the presence of standardized component catalogs outlining their functional and non-functional attributes.

At present, a software developer who uses the component-based approach cannot enjoy the same luxury. This is mainly because a majority of Commercial Off The Shelf (COTS) components are specified only with functional attributes in their interfaces. Typically, no concrete notion of quality is associated with components. Hence, the system developer has no means to objectively compare the performance characteristics of multiple components with the same functionality. This tends to restrict the developer's options when trying to select a component with a given functionality during the software development process. Thus, there is a need for a framework that would allow objective measurements of a component's Quality of Service (QoS) attributes. The creation of a Quality of Service catalog for software components would be the first step in this direction. Such a catalog should contain detailed descriptions about QoS attributes of software components along with the appropriate metrics, evaluation methodologies and the interrelationships with other attributes.

As a part of the UniFrame project [1], we are creating a Quality of Service-based framework for distributed heterogeneous software components. It is expected that this framework would initiate a standardization process in the component-based software development community. This would prove to be beneficial to the COTS component developer (producer) and the system developer (consumer). It would enable the component developer to advertise the quality of his components by using the QoS metrics, and allow the system developer to verify and validate the claims of the component developer.

The rest of the paper is organized as follows. The next section contains a discussion about work related to QoS in other domains like networking and in the domain of software. In section 3, the QoS framework is described in detail, along with a brief description of the UniFrame project. In section 4, as an application of the QoS framework, a detailed case study is presented from the domain of banking. An outline of our future plans is presented in section 5. Finally, we conclude in section 6.

2. Related Work:

The notion of QoS has been largely associated with the field of networking. A number of architectures have been proposed for QoS guarantees for distributed multimedia systems. In [2], a quality of service architecture (QoS-A) to

[1]Department of Computer and Information Science, Indiana University Purdue University Indianapolis, {gbrahnma, rraje, aolson}@cs.iupui.edu; [2]Computer Science Department, Naval Post Graduate School (on leave from New Mexico State University) auguston@cs.nps.navy.mil; [3]Department of Computer and Information Sciences, The University of Alabama at Birmingham, {bryant, cburt}@cis.uab.edu.

specify and achieve the necessary performance properties of continuous media applications over asynchronous transfer mode (ATM) networks is proposed. In QoS-A, instead of considering the QoS in the end-system and the network separately, a new integrated approach, which incorporates QoS interfaces, control, and management mechanisms across all architectural layers, is used. This architecture is based on the notions of flow, service contract and flow management. A service contract makes it possible to formalize the QoS requirements of the user and the potential degree of service commitment of the service provider. It also enables the specification of the network resource requirements and the necessary actions to be taken in case of a service contract violation. Flow management is utilized to monitor and maintain the QoS specified in the service contract.

The Quality Objects (QuO) framework [3] provides QoS to distributed software applications composed of objects. QuO is intended to bridge the gap between the socket-level QoS and the distributed object level QoS. This work mainly emphasizes on specification, measurement, control and adaptation to changes in quality of service. QuO extends the CORBA functional IDL with a QoS description language (QDL). QDL is a suite of quality description languages for describing QoS contracts between clients and objects, the system resources and mechanisms for measuring and providing QoS and adaptive behavior on the client and object side. It utilizes the Aspect Oriented Programming paradigm [4], which provides support for incorporating the non-functional properties of components separately from the functional properties.

QoS Modeling Language (QML) is a QoS specification Language proposed in [5]. QML is an extension of UML. It is a general purpose QoS specification language capable of describing different QoS attributes in any application domain. If offers three main abstraction mechanisms for QoS specification: contract type, contract and profile. A contract type represents a specific QoS attribute like: reliability or performance and it defines dimensions that can be used to characterize a particular QoS attribute. A contract is defined as an instance of a contract type and it represents a particular QoS specification. Profiles are used to associate contracts with interface entities such as operations, operation arguments and operation results. Here, the QoS specifications are syntactically separate from interface definitions, allowing different implementations of the same service interface to have different QoS characteristics. Thus a service specification may comprise of a functional interface and one or more QoS specifications.

The following features of the UniFrame approach, for QoS, distinguish it from other related efforts:

1. A creation of a QoS Catalog for software components containing detailed descriptions about QoS attributes of software components including the metrics, evaluation methodologies and the interrelationships with other attributes.
2. An integration of QoS at the individual component and distributed system levels.
3. A formal specification, based on Two-Level Grammars (TLG) [6], of the QoS attributes of each component.
4. The validation and assurance of QoS, based on the concept of event grammars [7].
5. An investigation of the effects of component composition on QoS; involving the estimation of the QoS of an ensemble of software components given the QoS of individual components.
6. A QoS-centric iterative component-based software development process, to ensure that the end-product matches both the functional and QoS specifications.

In this paper, we have addressed only the first two features. The details of the other features are discussed in [1].

3. QoS Framework for Software components:

3.1 UniFrame Project:

Our work on the QoS framework is part of the Unified Meta Component Model Framework (UniFrame) project. The UniFrame research attempts to unify the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques. This research targets not only the dynamic assembly of distributed software systems from components built using different component models, but also the necessary instrumentation to enable QoS features of the component and the ensemble of components to be measured and validated. The core parts of UniFrame project are: components, service and service guarantees and infrastructure.

Component: In UniFrame, components are autonomous entities, whose implementations are non-uniform, i.e.; each component adheres to a distributed-component model but there is no notion of a unified implementation framework. Each component has a state, an identity, a behavior, a well-defined interface and a private implementation.

Service and Service Guarantees: A service offered by a component could be an intensive computational effort or an access to underlying resources. In a DCS, it is natural to expect several choices for obtaining a specific service. Thus, each component must be able to specify the quality of service (QoS) offered. The QoS is an indication given by a component, on behalf of its owner, about its confidence to carry out the required services. The QoS offered by each component depends upon the computation it performs, the algorithm used, its expected computational effort, required resources, the motivation of the developer, and the dynamics of supply and demand.

Infrastructure: The headhunter and Internet Component Broker are responsible for allowing a seamless integration of different component models and sustaining cooperation among heterogeneous components. The tasks of headhunters are to detect the presence of new components in the search space, register their functionalities, and attempt matchmaking between service producers and consumers. It attempts at discovering components and registering them. Headhunters may cooperate with each other in order to serve a large number of components. The Internet Component Broker (ICB) acts as a translator between heterogeneous components. Adapter components register with ICB and indicate their specializations (which component models they can bridge efficiently). During a request from a seeker, the headhunter component not only searches for a provider, but also supplies the necessary details of an ICB.

Automated System Generation: In general, different developers will provide on the Internet a variety of possibly heterogeneous components oriented towards a specific problem domain. Once all the components necessary for implementing a specified distributed system are available and specific problem is formulated, then the task is to assemble them into a solution. UniFrame takes a pragmatic approach, based on Generative Programming [8,9], to component-based programming. It is assumed that the generation environment will be built around a generative domain-specific model (GDM) supporting component-based system assembly.

Further details about the UniFrame project can found in [1] [10] and [11].

3.2 Objectives of QoS Framework:

The QoS framework is a critical part of the UniFrame approach. The objectives of the QoS Framework are:

a) Identification of QoS attributes: A software component may be used in many different domains. Every domain has its own constraints with respect to the QoS attributes of software components. Hence, it is necessary to prepare a comprehensive compilation of different QoS attributes for many domains in which a software component may be used. Such a compilation would act as a checklist for any component developer/user interested in identifying the QoS attributes of interest.

b) Classification of QoS attributes based on:

   i. *Domain of usage:* Such a classification would enable a component user to identify the attributes that are relevant to his/her domain.

   ii. *Static / Dynamic behavior:* Such a classification would be helpful to determine whether a value of a QoS attribute is constant or varies according to the environment. This would in turn help in determining whether the value of a QoS attribute can be improved by changes to the operating environment.

   iii. *Nature of the attribute:* The QoS attributes identified are classified according to their characteristics into: *Time-related attributes (end-to-end-delay, freshness), Importance–related attributes (priority, precedence), Performance-related attributes (throughput, capacity), Integrity-related attributes (accuracy), Safety-related attributes (security) and Auxiliary attributes (portability, maintainability).*

   iv. *Composability of the attributes:* This kind of classification is important when different components are integrated to form a software system. It indicates whether the value of a given QoS attribute can be used in computing the value of the corresponding QoS attribute of the resultant system. Some of the QoS attributes are inherently non-composable, for example, parallelism constraints, priority, ordering constraints, etc. Hence, this classification would be valuable during the system integration phase.

c) Identification of metrics for QoS attributes: QoS metrics are the units for measuring the QoS attributes of a software component. Quantification of the QoS attributes of software components is one of the important goals of the proposed QoS framework. Hence, there is a need for standardized metrics to compare the QoS attributes of different software components. This would help to ensure uniformity in the expression of the QoS attributes.

d) Creation of a QoS catalog for Software Components: The QoS Catalog would act as a comprehensive source of information regarding the quality of software components. It would contain detailed descriptions about QoS attributes of software components including the metrics, evaluation methodologies and the interrelationships among the QoS attributes.

e) Creation of a QoS interface for a component with different levels of details: One of the primary objectives of the QoS framework is to make the QoS attributes an integral part of a software component. The QoS interface is aimed at achieving this objective. The QoS interface would contain the values for QoS attributes of a software component.

For the sake of brevity, here, only the concepts of QoS parameters and the QoS catalog are discussed.

3.3 Catalog of QoS Parameters:

The QoS Catalog for Software components would prove to be a valuable tool for:

i. The component developer by: a) acting as a reference manual for incorporating QoS attributes into the components being developed, b) allowing him to enhance the performance of his component in an iterative fashion by being able to quantify their QoS attributes, and c) enabling him to advertise the Quality of his components using the QoS metrics.

ii. The system developer by: a) enabling him to specify the QoS requirements of the components that are incorporated into his system, b) allowing him to verify and validate the claims of the component developer, c) allowing him to make objective comparisons of QoS of components having the same functionality, and d) empowering him with the means to choose the best suited components for his system.

At present the following QoS parameters have been selected for inclusion in the catalog. More parameters will be included as they are identified.

1. *Dependability:* It is a measure of confidence that the component is free from errors.
2. *Security:* It is a measure of the ability of the component to resist an intrusion.
3. *Adaptability:* It is a measure of the ability of the component to tolerate changes in resources and user requirements.
4. *Maintainability:* It is a measure of the ease with which a software system can be maintained.
5. *Portability:* It is a measure of the ease with which a component can be migrated to a new environment.
6. *Throughput:* It indicates the efficiency or speed of a component.
7. *Capacity:* It indicates the maximum number of concurrent requests a component can serve.
8. *Turn-around Time:* It is a measure of the time taken by the component to return the result.
9. *Parallelism Constraints:* It indicates whether a component can support synchronous or asynchronous invocations.
10. *Availability:* It indicates the duration when a component is available to offer a particular service.
11. *Ordering Constraints:* It indicates the order of returned results and its significance.
12. *Evolvability:* It indicates how easily a component can evolve over a span of time.
13. *Result:* Indicates the quality of the results returned.
14. *Achievability:* It indicates whether the component can provide a higher degree of service than promised.
15. *Priority:* It indicates if a component is capable of providing prioritized service.
16. *Presentation:* It indicates the quality of presentation of the results returned by the component.

Detailed sample descriptions of two of the above-mentioned QoS parameters, Dependability and Turn-around Time, are given below:

Name:	**DEPENDABILITY**
Intent:	It is a measure of confidence that the component is free from errors.
Description:	It is defined as the probability that the component is defect free.
Motivation:	1. It allows an evaluation of degree of Dependability of a given component. 2. It allows a comparison of Dependability of different components. 3. It allows for modifications to a component to increase its Dependability.
Applicability:	This parameter can be used in any system, which requires its components to offer a specific level of dependability. Using this parameter, the Dependability of a given component can be calculated before being incorporated into the system.
Model Used:	Dependability model by J. Voas and J. Payne [12].
Metrics used:	Testability Score, Dependability Score.
Influencing Factors:	1. Degree of testing. 2. Fault hiding ability of the code. 3. The likelihood that a statement in a component is executed. 4. The likelihood that a mutated statement will infect the component's state. 5. The likelihood that a corrupted state will propagate and cause the component output to be mutated.
Evaluation Procedure:	1. Perform Execution Analysis on the component. 2. Perform Propagation Analysis on the component. 3. Calculate the Testability value of the component.

	4. Calculate the Dependability Score of the component.
Evaluation Formulae:	$T = E * P$. *T: Testability Score (a prediction of the likelihood that a particular statement in a component will hide a defect during testing).* *E: Execution Estimate (the likelihood of executing a given fault).* *P: Propagation Estimate (the conditional probability of the corrupted data state corrupting the software's output after the state gets infected).* $D = 1-(1-T)^N$. *D: Dependability Score.* *N: Number of successful tests.*
Result Type:	Floating Point Value between [0,1].
Static / Dynamic:	Static.
Composable / Non-Composable:	Composable.
Consequence:	1. Greater amounts of testing and greater Testability scores result in greater Dependability. 2. Lesser amount of testing is required to provide a fixed dependability score for higher Testability Scores.
Related Parameters:	Availability, Error Rate, Stability.
Domain of Usage:	Domain Independent.
Error Situation:	Low dependability results in: 1. Unreliable component behavior. 2. Improper execution / termination. 3. Erroneous results.
Aliases:	Maturity, Fault Hiding Ability, Degree of Testing.

Name:	**Turn-around Time**
Intent:	It is a measure of the time taken by the component to return the result.
Description:	It is defined as the time interval between the instant the component receives a request until the final result is generated.
Motivation:	1. It indicates the delay involved in getting results from a component. 2. It is one of the measures of the performance offered by a component.
Applicability:	This attribute can be used in any system, which specifies bounds on the response times of its components.
Model Used:	Empirical approach.
Metrics Used:	Mean Turn-around Time.
Influencing Factors:	1. Implementation (algorithm used, multi-thread mechanism etc). 2. Speed of the CPU. 3. Available memory. 4. Load on the system. 5. Operating System's access policy for resources like: CPU, I/O, memory, etc.
Evaluation Procedure:	1. Record the time instant at which the request is received. 2. Record the time instant at which the final result is produced. 3. Repeat steps 1 and 2 for 'n' representative requests. 4. Calculate the Mean Turn-around Time.
Evaluation Formulae:	$MTAT= [\Sigma_{i=1}^{n} (t2-t1)] / n$. MTAT: Mean Turn-around Time. t1: time instant at which the request is received. t2: time instant at which the final result is produced. n: number of representative requests.
Result Type:	Floating Point Value in milliseconds.
Static / Dynamic:	Dynamic.
Composable / Non-Composable	Composable.

Consequence:	Lower the time interval between the instant the request is received and the response is generated, lower the Mean Turn-around Time.
Related Parameters:	Throughput, Capacity.
Domain of Usage:	Domain Independent.
Error Situation:	A high value of Internal Response Time results in: 1. Longer delays in producing the result. 2. Higher round trip time.
Aliases:	Latency, Delay.

4. Case Study:

Let us assume that a private bank is trying to build a software system to automate its day-to-day operations. The bank has decided to utilize a Client-server Distributed computing model .The bank has also chosen to assemble the system using COTS software components instead of building the system from scratch.

The In-house software development team in the bank has come out with the following simple design for the system:

- The system consists of two categories of components: AccountServer and AccountClient.
- There will be two instances of the AccountServer and one instance of the AccountClient.
- The two AccountServers are of type javaAccountServer, adhering to the java-RMI model and corbaAccountServer, adhering to the CORBA model.
- The components should offer the following functionality: Deposit, Withdraw and Balance check

The system development team now needs three different components meeting the above functionality requirements. However, the bank also expects the components to satisfy certain QoS requirements. These are listed below:

- Dependability: The components will be an integral part of the bank and be responsible for keeping track of all transactions within the bank. Hence the component should offer some guarantees regarding error free operation.
- Turn-around Time: The transactions within the banking system have time restrictions imposed on them. Hence, they have to produce results within a specified time frame. This requires that the components satisfy Turn-around time requirements.

The partial UniFrame descriptions of these components are presented below:

JavaAccountServer:	**CorbaAccountServer:**
Informal Description: Provides an account management service. Supports three functions: javaDeposit(), javaWithdraw() and javaBalance().	Informal Description: Provides an account management service. Supports three functions: corbaDeposit(), corbaWithdraw() and corbaBalance().
1. Computational Attributes:	1. Computational Attributes:
a) Inherent Attributes:	a) Inherent Attributes:
a.1 id: intrepid.cs.iupui.edu/jServer	a.1 id: jovis.cs.iupui.edu/coServer
b) Functional Attributes:	b) Functional Attributes:
b.1 Acts as an account server	b.1 Acts as an account server
b.2 Algorithm: simple addition/subtraction	b.2 Algorithm: simple addition/subtraction
b.3 Complexity: O(1)	b.3 Complexity: O(1)
b.4 Syntactic Contract:	b.4 Syntactic Contract:
void javaDeposit(float ip);	void corbaDeposit(float ip);
void javaWithdraw(float ip) throws overDrawException;	void corbaWithdraw(float ip) throws overDrawException;
float javaBalance();	float corbaBalance();
b.5 Technology: Java-RMI	b.5 Technology: Java-CORBA
....	....
2. Cooperation Attributes:	2. Cooperation Attributes:
2.1) Pre-processing Collaborators: AccountClient	2.1) Pre-processing Collaborators: AccountClient
3. Auxiliary Attributes:	3. Auxiliary Attributes:
....	....
4. QoS Metrics:	4. QoS Metrics:
Dependability = 0.98	Dependability = 0.99
Turn-around Time: MTAT=70	Turn-around Time: MTAT=80

JavaAccountClient:
Informal Description: Requests account services from an appropriate server and interacts with the user; implemented as a web-based applet. Supports functions: depositMoney(), withdrawMoney() and checkBalance().

1. Computational Attributes:
a) Inherent Attributes:
a.1 id: galileo.cs.iupui.edu/aClient
 	
b) Functional Attributes:
b.1 accepts user queries and presents the results using a GUI
b.2 Algorithm: Java Foundation Classes (JFC)
b.3 Complexity: O(1)
b.4 Syntactic Contract
void depositMoney(float ip);
void withdrawMoney(float ip);
float checkBalance();
b.5 Technology: Java Applet
....
	2. Cooperation Attributes:
	2.1) Post-processing Collaborators: AccountServer
	
	3. Auxiliary Attributes:
	....
	
	4. QoS Metrics:
	Dependability = 0.99
	Turn-around Time: MTAT = 90

Query and Returned Results:
A sample query for the above example can be informally stated as: Create an account management system that has: Dependability > 0.97 and Turn-around Time: MTAT < 100. From the query and the available knowledge in the GDM associated with the account management systems, a formal specification of the desired system will be formulated for a headhunter in UniFrame. In response, the headhunter will discover the following choices:

1. Java-Java System: a) javaAccountClient -- Dependability = 0.99, Turn-around Time: MTAT = 90, Java Applet Technology b) javaAccountServer -- Dependability = 0.98, Turn-around Time: MTAT = 70, Java-RMI technology c) Infrastructure Needed -- JVM and Appletviewer.

2. Java-CORBA System: a) javaAccountClient -- Dependability = 0.99,Turn-around Time: MTAT= 90, Java Applet Technology b) corbaAccountServer – Dependability = 0.99,Turn-around Time: MTAT= 80, Java-RMI technology c) Infrastructure Needed -- JVM, Appletviewer, ORB, Java-CORBA bridge.

QoS of the assembled system:
Each component has two QoS parameters: 1) static – dependability and 2) dynamic - Turn-around Time. The desired QoS of the assembled system includes these parameters as well. For this reason the GDM will contain a rule that will compute the value of the static parameter for the assembled system. In this example, the dependability for the assembled system is calculated using the following formula: $(1.0 - ((1.0 - D_1) + (1.0 - D_2))$, Where, D_1 and D_2 are the dependability values of the constituent components, yielding a value of 0.97 for the Java-Java System and a value of 0.98 for the Java-CORBA System.

For the dynamic parameter, the generator will provide the necessary instrumentation for taking the clock and calculating the Turn-around Time at run-time. The knowledge about metrics for the QoS parameter ` Turnaround Time' is represented in terms of Duration attribute for events of the type method -call, and the generic computation over the event trace that takes the clock and sums up those durations yielding a measured Turn-around Time for the accounting system.

One of the two example systems, mentioned in the query, will be implemented with the code for carrying out event trace computations according to user-supplied test cases. These test cases will be executed to verify that the accounting system satisfies the QoS specified in the query. If the system is not verified, it is discarded. This verification process is carried out for each of the generated accounting systems (two in the above example). Then, the one with the best QoS is chosen.

5. Future Plans:
Incorporation of the above-mentioned QoS parameters into the component interface is our next step. This would involve the creation of a QoS interface of the component along the lines of a functional (or syntactical) interface of a

component. This QoS interface would include all the necessary information about those QoS parameters that are selected by the component developer for inclusion in a given component. This would be followed by a formal specification of these QoS parameters and a mechanism for ensuring them at the individual component level and at the system level. The issue of Quality of Service of an ensemble of software components, i.e., a software system built out of components would also be addressed. This would involve the issues of component composition and composability of QoS Parameters.

6. Conclusion:

This paper has presented a QoS framework for software components, which is a part of the UniFrame project [1]. The objectives of the QoS framework include: a) the creation of a QoS catalog designed to quantify the QoS attributes of software components, b) incorporation of QoS attributes into the component interface, c) a formal specification of these attributes, d) a mechanism for ensuring these attributes at individual component level and at the system level, and e) a procedure to estimate the QoS of an ensemble of software components. Due to the space restrictions, only the concepts of QoS parameters and QoS catalog are presented here. The QoS framework would enable the component developer to advertise the quality of his components by using the QoS metrics, and allow the system developer to verify and validate the claims of the component developer. Although a simple case study is provided in this paper, the principles of the proposed approach are general enough to be applied to any larger applications.

Acknowledgments: The material presented in this paper is based upon work supported by, or in part by, a) the U.S. Office of Naval Research under award number N00014-01-1-0746, b) the U.S. Army Research Laboratory and the U.S. Army Research Office under contract/grant number 40473-MA.

7. References:

1) R. Raje, M. Auguston, B. Bryant, A. Olson, C. Burt. *A Quality of Service – based framework for creating distributed heterogeneous software components*, Technical Report, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2001.

2) A. Campbell. *A Quality of Service Architecture* –Ph.D. Thesis, Computing Department, Lancaster University, 1996.

3) BBN Corporation, Quality Objects Project, URL: *http://www.dist-systems.bbn.com/tech/QuO,* 2001.

4) Communications of ACM special issue on Aspect Oriented Programming, vol.44, No 10, October 2001.

5) S. Frolund, J. Koistinen. *Quality of Service specification in distributed object systems*, Distributed System Engineering Journal, Vol.5, No. 4, December, 1998

6) A. Van Wijngaarden. *Orthogonal Design and Description of a formal Language*. Technical Report, Mathematisch Centrum, Amsterdam, 1965.

7) M. Auguston. Program *Behavior Model Based on Event Grammar and it's Application for Debugging Automation.* In Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, 1995.

8) Batory, D. and Chen, G. and Robertson, E. and Wang, T. *Design Wizards and Visual Programming Environments for Gen Voca Generators.* IEEE Transactions on Software Engineering, pages 441-452, 2000.

9) Czarski, K., and Eisenecker, U.W. *Generative Programming: Methods, Tools, and Applications.* Addison - Wesley, 2000.

10) R. Raje. *"UMM: Unified Meta-object Model for Open Distributed Systems"*, Proceedings of the fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing pages 454-465 (ICA3PP' 2000).

11) R. Raje, M. Auguston, B. Bryant, A. Olson, C. Burt, *"A Unified Approach for the Integration of Distributed Heterogeneous Software Components"*, Proceedings of the 2001 Monterey Workshop (Sponsored by DARPA, ONR, ARO and AFOSR), Monterey, California, 2001.

12) J. Voas, J. Payne, *Dependability Certification of Software Components*, Journal of Systems and Software, NO. 52, pp. 165-172, 2000.

SESEC2002
April 2002

Quality of Service (QoS) Standards for Model Driven Architecture[1]

Carol C. Burt[2], Barrett R. Bryant[2], Rajeev R. Raje[3], Andrew Olson[3], Mikhail Auguston[4]

Abstract

A number of middleware technologies have evolved over the last ten years to address specific business problems such as enabling process optimization via systems integration, rapid development of new applications, web enabling features for customers, and mechanization of supply chains. Software architects increasingly utilize models to represent different viewpoints of a business solution. Separation of concerns is a key characteristic of good software design. In an effort to facilitate the design of business systems in a platform independent matter, the Object Management Group (OMG) is currently progressing the Model Driven Architecture (MDA)[1]. Model Driven Architecture maintains a clean separation of Platform Independent Models (PIMs) that represent the domain from Platform Specific Models (PSMs) that expose details related to a middleware technology. In this way a single PIM can be mapped to multiple implementation technologies (such as OMG CORBA®, Sun J2EE, Microsoft COM+, and W3C Web Services). While all of the component-based technologies have established a concise means for describing functional contracts (the interface or services offered by a component in the architecture), none of these technologies have embraced architecture or a vocabulary for specifying non-functional Quality of Service (QoS) contracts. Non-functional contracts are necessary to analyze the ability of a service to meet QoS constraints when used in a composition. A component may, for example, guarantee a certain performance level (given a fixed set of constraints) or guarantee protection of features and/or information classified as a protected resource. The UniFrame [2] research (which includes identification and progression of requisite standards activities) envisions a plug and play component environment where QoS contracts are part of a component description and middleware bridges and quality of service instrumentation are generated by component integration toolkits. That is, business components that utilize diverse platform technologies may be easily integrated, and they will offer both functional and non-functional service contracts. A difficulty in progressing this work is the lack of a standard vocabulary for software component Quality of Service (QoS). Following this work, syntax for expressing QoS in component models and the mappings that form the transformations from diverse viewpoints of a model must also be explored and standardized. This research includes supporting and participating in the progression of such standards. This paper introduces a path for standards in the area of Quality of Service and discusses how this standardization would progress the goal of using commercial off the shelf (COTS) components in a heterogeneous system composition.

---

[1] This research is supported by U. S. Office of Naval Research award N00014-01-1-0746.
[2] Department of Computer and Information Sciences, The University of Alabama at Birmingham, Birmingham, AL 35294, USA, {cburt, bryant} @cis.uab.edu
[3] Department of Computer and Information Science, Indiana University Purdue University Indianapolis, Indianapolis, IN 46202, USA, {rraje, aolson} @cs.iupui.edu
[4] Computer Science Department, New Mexico State University, Las Cruces, NM 88003, USA, mikau@cs.nmsu.edu

Introduction

Enterprises are increasingly dependent upon multiple middleware technologies that enable new business paradigms by weaving together legacy systems with advanced technology. These technologies support core business functionality, enable distributed business systems, integrate business processes and allow companies to communicate with customers, suppliers, and business partners. While it is possible to construct such systems, it requires that the developer be aware of the nuances of the diverse middleware technologies. This problem must be resolved for the promise of software component technology (plug & play / off the shelf interoperability) can be fully realized. In addition, the increased complexity of this environment makes it impossible to predict the non-functional aspects of such a system until after it is constructed. That is, static QoS relies on the design expertise of software architects and engineers and metrics and test scenarios must be hand crafted on a case-by-case basis to determine if a composition is acceptable. As distributed systems become omni-present with many mission-critical, the notion of QoS-oriented software development will become essential. Such an approach is necessary to ensure the reliability and high confidence of distributed software systems.

The Unified Component Meta Model Framework (UniFrame) [2] research project is an attempt to unify the existing and emerging distributed component models under a common meta-model for the purpose of enabling the discovery, interoperability, and collaboration of components via generative software techniques. A great deal of work is underway in standards organizations such as OMG and World Wide Web Consortium (W3C) that provides the foundation for UniFrame. The UniFrame research builds upon work in standards organizations, targeting the dynamic assembly of distributed software systems from components built in different component models and the ability to express quality of service (QoS) requirements in such a way that generative design and implementations can utilize them. This is a necessity to enable timely (perhaps dynamic) assembly of e-business relationships (for example to select a replacement component when a primary component fails to deliver on QoS guarantees). It is also necessary in time and safety critical environments where failure to meet quality of service requirements results in significant system failures.

Related Work

Although QoS parameters and associated metrics have been widely used in networking, there is no standard vocabulary for discussing QoS as it relates to distributed computing and component based solutions. For example, the CORBA® Components Specification only uses the term "quality of service" with regard to events and whether or not they are transactional in nature [3]. The Java2 Enterprise Edition (J2EE) specification [4] states, "We expect J2EE products to vary widely and compete vigorously on various aspects of quality of service. Products will provide different levels of performance, scalability, robustness, availability, and security. In some cases this specification requires minimal levels of service. Future versions of this specification may allow applications to describe their requirements in these areas." Today, there is no standard vocabulary that we can utilize to define QoS requirements for any component technology.

In the fall of 2000, the OMG Analysis and Design Task Force considered a draft Request for Proposal (RFP) for a UML profile for Modeling Quality of Service as it relates to real-time systems [5]. This RFP called for a framework and categorization of QoS characteristics. In January 2002, the OMG Analysis and Design task force issued an RFP for a "UML™ Profile for

Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" [6]. This RFP solicits proposals for a UML profile or Meta Object Facility (MOF) meta-model that defines standard paradigms of use in modeling quality of service and fault-tolerance aspects of systems. This is the first of a series of RFPs that have the goal of significant benefits to the UML user community engaged in high-quality robust system development. The key mandatory requirements of this RFP are listed in Figure 1.

A General Quality of Service Framework

To ensure consistency in modeling various qualities of service, submissions shall define a standard framework or, reference model, for QoS modeling in the context of the UML. This shall include:

- A general categorization of different kinds of QoS; including QoS that are fixed at design time as well as ones that are managed dynamically
- Integration of different categories of QoS for the purpose of QoS modeling of system aspects.
- Identification of the basic conceptual elements involved in QoS and their mutual relationships. This shall include the ability to associate QoS characteristics to model elements (specification), a generic model of the system aspects involved in QoS-associated collaboration and their functional interactions and use cases (usage model), and a generic model of how QoS allocation and decomposition is managed.
- A coherent set of stereotypes, tagged values, and constraints as necessary to represent the identified QoS properties constructing a UML Profile.

A Definition of Individual QoS Characteristics

Submissions shall define QoS characteristics, particularly those important to real-time and high confidence systems, which describe the fundamental aspects of the various specific kinds of QoS based on the QoS categorization identified in the framework. These shall include but are not limited to the following:

- time-related characteristics (delays, freshness)
- importance-related characteristics (priority, precedence)
- capacity-related characteristics (throughput, capacity)
- integrity related characteristics (accuracy)
- fault tolerance characteristics (mean-time between failures, mean-time to repair, number of replicas)

A coherent set of stereotypes, tagged values, and constraints as necessary to represent the identified QoS properties constructing a UML Profile.

Figure 1: OMG RFP – UML Profile for QoS - Mandatory Requirements

As part of the UniFrame research, we have outlined our approach to a QoS-based framework for creating distributed heterogeneous software components [7]. The QoS-based method in UniFrame is made up of three steps:

1. The creation of a catalog for QoS parameters (and/or metrics),
2. A formal specification of these parameters, and
3. A mechanism for ensuring these parameters, both at each individual component level and at the entire system level

UniFrame leverages work by Zinky, Bakken & Schantz [8] with the goal of providing a catalog of QoS parameters, indicating how parameters might be described. There are many possible QoS parameters that a component (and its developer) can use to indicate the associated service. Some of these parameters may be general in nature, while others may be pertain to a specific domain. The goal of creating the QoS catalog is two fold: a) it assists the component developer (or the system integrator) in selecting the necessary QoS parameters for the component (or system) under construction, and b) it enables the developer (or integrator) to ensure the necessary QoS guarantees by integrating the selected QoS parameters into the assurance process.

We have recently published the initial version of our QoS catalog [9]. At present the following QoS parameters have been selected for inclusion in the catalog.

1. *Dependability:* a measure of confidence that the component is free from errors.
2. *Security:* a measure of the ability of the component to resist an intrusion.
3. *Adaptability:* a measure of the ability of the component to tolerate changes in resources and user requirements.
4. *Maintainability:* a measure of the ease with which a software system can be maintained.
5. *Portability:* a measure of the ease with which a component can be migrated to a new environment.
6. *Throughput:* indicates the efficiency or speed of a component.
7. *Capacity:* indicates the maximum number of concurrent requests a component can serve.
8. *Turn-around Time:* a measure of the time taken by the component to return the result.
9. *Parallelism Constraints:* indicates whether a component can support synchronous or asynchronous invocations.
10. *Availability:* indicates the duration when a component is available to offer a particular service.
11. *Ordering Constraints:* indicates the order of returned results and its significance.
12. *Evolvability:* indicates how easily a component can evolve over a span of time.
13. *Result:* indicates the quality of the results returned.
14. *Achievability:* indicates whether the component can provide a higher degree of service than promised.
15. *Priority:* indicates if a component is capable of providing prioritized service.
16. *Presentation:* indicates the quality of presentation of the results returned by the component.

Uniframe also leverages work on Quality Objects and adaptive middleware. Quality Objects [10] is a framework for providing QoS to software applications composed of objects distributed over wide area networks. QuO bridges the gap between socket-level QoS and distributed object level QoS, emphasizing specification, measuring, controlling, and adapting to changes in QoS. RAPIDware [11] is an approach to component-based development of adaptable and dependable middleware. It uses rigorous software development methods to support interactive applications executed across heterogeneous networked environments.

Frolund & Koistinen [12] point out that deciding which quality of service properties should be provided by individual components is an important part of the design process. They define a Quality-of-Service specification language (QML) and they show how the Unified Modeling Language (UML) can be extended to support the concepts of QML. In addition, they suggest a technique for representation of QML constructs in terms of ISO IDL [12] [13]. Frolund and Koistinen were working with a platform specific model (CORBA). The OMG currently is progressing an RFP for a UML profile for Quality of Service that will provide the meta-model necessary to use UML to model QoS in a platform independent manner.

The OMG work will standardize how static (design related) and dynamic (environmentally influenced) QoS characteristics are expressed in UML models. We are working within the OMG community to introduce our efforts, contribute to the analysis of the submissions, and ensure that our research is aligned with industry standard vocabulary as we progress techniques that enable QoS-aware systems to utilize generative software tools. We will also be experimenting with alternative syntax for representation of QoS characteristics such as event grammar [14].

Model Driven Architecture with QoS parameters

In addition to the work that OMG has done with distributed computing interoperability (CORBA®/IIOP), the OMG has also progressed standards in the domain of modeling and meta-modeling: the Unified Modeling Language (UML™) and Meta-Object Facility (MOF™). Some of the analysis and design standards include the precise mappings that define the transformation of model information into interface definitions in ISO Interface Definition Language (IDL).

The latest initiative - Model Driven Architecture (MDA™) - is the way that the OMG will standardize Platform Independent Models (PIMs) that can be mapped to multiple Platform Specific Models such as CORBA®, Java2 Enterprise Edition (J2EE), and Web Services for implementation. This approach holds promise for the standardization of components that could be used in collaborative environments as a result of a common semantic model. To fully realize the potential of this approach, Quality of Service (QoS) catalogs, formal parameterization of Platform Specific Models, and ultimately instrumentation mappings must also be standardized within the Model Driven Architecture roadmap.

Figure 2 outlines the type of models that are common in a MDA approach. Quality of Service parameters must be introduced into each model and the transformations (or mappings) that occur as models are refined must be standardized. The current RFP is merely the beginning – providing a vocabulary and syntax for expressing QoS in UML. As we move beyond the QoS catalog, our research will focus on the constraints that are placed on transformations as a result of the quality requirements and the generative techniques for ensuring that metrics can be gathered.

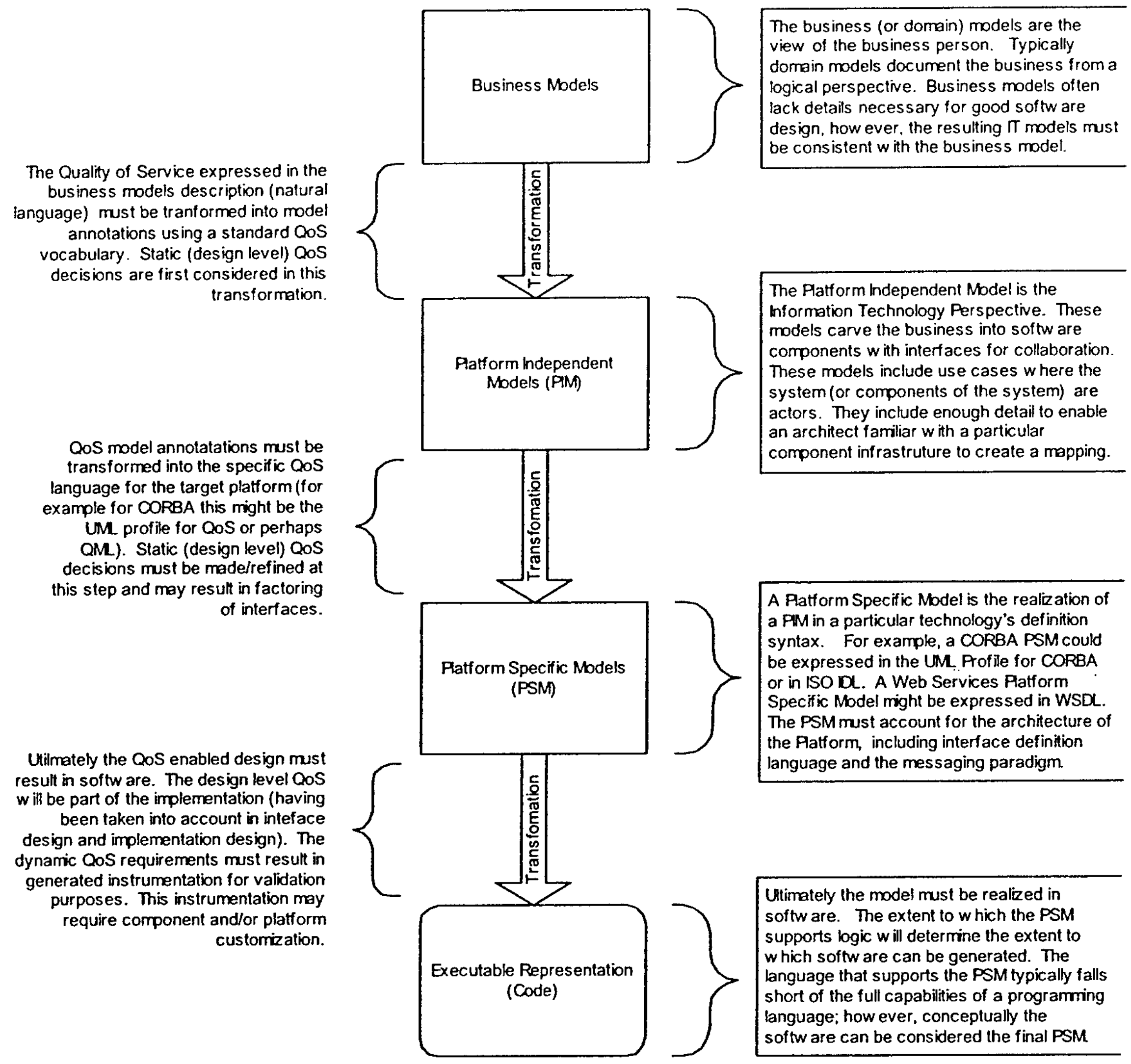

Business Models
The business (or domain) models are the view of the business person. Typically domain models document the business from a logical perspective. Business models often lack details necessary for good software design, however, the resulting IT models must be consistent with the business model.
Transformation
The Quality of Service expressed in the business models description (natural language) must be tranformed into model annotations using a standard QoS vocabulary. Static (design level) QoS decisions are first considered in this transformation.
Platform Independent Models (PIM)
The Platform Independent Model is the Information Technology Perspective. These models carve the business into software components with interfaces for collaboration. These models include use cases where the system (or components of the system) are actors. They include enough detail to enable an architect familiar with a particular component infrastruture to create a mapping.
Transformation
QoS model annotatations must be transformed into the specific QoS language for the target platform (for example for CORBA this might be the UML profile for QoS or perhaps QML). Static (design level) QoS decisions must be made/refined at this step and may result in factoring of interfaces.
Platform Specific Models (PSM)
A Platform Specific Model is the realization of a PIM in a particular technology's definition syntax. For example, a CORBA PSM could be expressed in the UML Profile for CORBA or in ISO IDL. A Web Services Platform Specific Model might be expressed in WSDL. The PSM must account for the architecture of the Platform, including interface definition language and the messaging paradigm.
Transformation
Utilmately the QoS enabled design must result in software. The design level QoS will be part of the implementation (having been taken into account in inteface design and implementation design). The dynamic QoS requirements must result in generated instrumentation for validation purposes. This instrumentation may require component and/or platform customization.
Executable Representation (Code)
Ultimately the model must be realized in software. The extent to which the PSM supports logic will determine the extent to which software can be generated. The language that supports the PSM typically falls short of the full capabilities of a programming language; however, conceptually the software can be considered the final PSM.

Figure 2 – QoS considerations during model transformations

Quality of Service Issues related to Interface Generation

It should be noted that there are existing standards for using generative techniques to create interfaces from UML models. For example, the OMG Meta Object Facility (MOF) [15] allows the generation of interfaces from Unified Modeling Language UML models. A careful analysis of the resulting interface specifications makes it clear, however, that distribution is not a key factor in the algorithms used. This has a direct impact on the ability to meet quality of service requirements in a distributed solution. For example, in a distributed system, quality of service requirements for performance, scalability and/or security would dictate the use of iterators, the factoring of interfaces into separate query and administrative operations and the use of structure and/or objects passed by value. The current standards in this area tend to focus on data access with accessors and mutators and relationship traversal. This is acceptable (perhaps even desirable) in a single machine environment, but unacceptable for highly distributed communications and collaborations [1]. It has long been accepted that systems with distributed components require specialized design to ensure performance and ease of security administration. That is, there is a need to take QoS parameters and use-cases into account when designing component interfaces. For a model driven generative technique to be successful, models must be parameterized with QoS standard parameters as defined in a catalog. In addition, use case scenarios must also be formally expressed so that they can be used as input to an interface generator. That is, given a parameterized domain model, semantically equivalent interfaces (and the bridges between them) must be generated.

The OMG Architecture Board produced a paper that describes the technical details of the Model Driven Architecture (MDA) [1]. This document outlines several areas where significant research is required before the MDA vision can be fully realized. One of the most important areas is directly related to the UniFrame research. It states: "It is generally agreed that the MOF-IDL mapping is in need of upgrading. [Footnote: The problem is that the generated interfaces are not efficient in distributed systems. Firstly, the mapping predates CORBA valuetypes and thus does not make use of them. Secondly, a class with N attributes is always mapped to a CORBA interface with N separate getter/setter operations. In a distributed system one would want to group attributes based upon use cases, cache attribute values, or implement other optimizations to reduce the number of distributed calls]. Realistically we will probably have to accept the fact that for the foreseeable future, the automatically generated transformation from PIM to PSM will have to be enhanced by humans. As we gain more experience we will be able to define various patterns and allow them to be selected in some way."

This is recognition that a generated interface must be optimized using quality of service and usage scenarios requires research in techniques for integrating QoS into a generative programming model. It also recognizes that we do not currently have a way to express the quality of service requirements in such a way that generative techniques can be trusted during the design process.

Conclusion

The ability to provide QoS parameterization of models is recognized in the Object Management Group community and standards in this area will lead to the ability to generate Platform Specific Models that take quality of service characteristics into account. Since there has been very little work on progressing Quality of Service specifications for component based architectures, this work has the potential to impact how the Object Management Group (OMG) defines QoS parameterization for Model Driven Architecture and the ability to more clearly specify and measure component feasibility for a particular task. This standardization of QoS catalogs and parameters is a pre-requisite to benchmarking and service validation instrumentation. In addition, the Java Community Process (JCP) has a history of working with OMG to progress consistent standards. The expectation is that any Quality of Service parameters would be applicable for CORBA®, J2EE™, and Web Services component architectures. In addition, this standardization provides a foundation for future standards. Quality of Service characteristics must have syntax for expression in every artifact of the analysis, design and development process. The design patterns must be documented and exploited in such a way that generative techniques can be applied. In addition, formal specifications will allow instrumentation necessary for measuring quality of service to be come an integral part of middleware and component implementation frameworks.

The UniFrame research project is investigating techniques and patterns used when static QoS is a consideration during refinement of models and software design, and is utilizing emerging technology in generative programming for QoS instrumentation with a goal of progressing standards for QoS instrumentation when the technology matures. By ensuring that software components can be tested against standard Quality of Service feature sets we progress the goal of using more commercial off the shelf (COTS) components in heterogeneous system compositions.

References

[1] Object Management Group. 2001. *Model Driven Architecture: A Technical Perspective.* Technical Report. Document # ormsc/2001-07-01. Framingham, MA: Object Management Group. July 2001.

[2] Rajeev R. Raje, Barrett Bryant, Mikhail Auguston, Andrew Olson, Carol Burt. "*A Unified Approach for the Integration of Distributed Heterogeneous Software Components*", Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration, pp: 109-119, Monterey, California, 2001.

[3] Object Management Group. 2001. *CORBA 3.0 CORBA Component Model Chapters.* Document # ptc/2001-11-03. Framingham, MA: Object Management Group.

[4] Sun Microsystems. 2001. *Java 2 Platform Enterprise Edition Specification v1.3*, Available via ftp from www.java.sun.com. Sun Microsystems.

[5] Object Management Group. 2000. *UML Profile for Modeling Quality of Service as it relates to real-time systems.* Draft Request for Proposal. OMG document ad/00-12-07. Framington, MA. Note: This RFP was never issued.

[6] Object Management Group. 2002. *UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms.* Request for Proposal. OMG document ad/02-01-07. Framington, MA. Note: This RFP issued January 2002 with submissions due June 24, 2002.

[7] Rajeev R. Raje, Mikhail Auguston, Barrett Bryant, Andrew Olson, Carol Burt. 2001. *A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components.* Technical Report. Indiana University Purdue University Indianapolis.

[8] Zinky, J.A.,Bakken, D.E., and Schantz, R., 1995. *Overview of Quality of Service for Distributed Objects,* In Proceedings of the Fifth IEEE Dual Use Conference.

[9] G. Brahnmath, R. Raje, A. Olson, M. Auguston, B. Bryant and C. Burt. 2002. *Quality of Service Catalog for Software Components.* Technical Report #TR-CIS-0219-01. Indiana University Purdue University Indianapolis.

[10] BBN Corporation, 2001. *Quality Objects (QuO) Project,* URL: http://www.dist-systems.bbn.com/tech/QuO.

[11] Michigan State University, 2001. *RAPIDware: Component-Based Development of Adaptable and Dependable Middleware,* URL: http://www.cse.msu.edu/rapidware/.

[12] S. Frolund, J. Koistinen. 1998. *Quality of Service specification in Distributed Object Systems*, Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98), Santa Fe, New Mexico, April 1998.

[13] S. Frolund, J. Koistinen. 1999. *Quality of Service Aware Distributed Object Systems*. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99). May 1999.

[14] M.Auguston, 1998. *Building Program Behavior Models*, Proceedings of the European Conference on Artificial Intelligence ECAI-98, Workshop on Spatial and Temporal Reasoning, Brighton, England, August 23-28, 1998, pp.19-26.

[15] Object Management Group. 2000. *Meta Object Facility*. Document formal/2001-11-02. Framingham, MA, Object Management Group.

An Architecture for the UniFrame Resource Discovery Service[1]

Nanditha N. Siram, Rajeev R. Raje, Andrew M. Olson
Department of Computer and Information Science
Indiana University Purdue University Indianapolis
723 W. Michigan Street, SL 280
Indianapolis, IN 46202-5132, USA
Email: {nnayani, rraje, aolson}@cs.iupui.edu

Barrett R. Bryant, Carol C. Burt
Department of Computer and Information Sciences
The University of Alabama at Birmingham
115A Campbell Hall, 1300 University Boulevard
Birmingham, AL 35294-1170, USA
Email: {bryant, cburt}@cis.uab.edu

Mikhail Auguston
Department of Computer Science
New Mexico State University
PO Box 30001, MCS CS
Las Cruces, NM 8803
Email: mikau@cs.nmsu.edu

Abstract

Frequently, the software development for large-scale distributed systems requires combining components that adhere to different object models. One solution for the integration of distributed and heterogeneous software components is the *UniFrame* approach. It provides a comprehensive framework unifying existing and emerging distributed component models under a common meta-model that enables the discovery, interoperability, and collaboration of components via generative software techniques. This paper presents the architecture for the resource discovery aspect of this framework, called the UniFrame Resource Discovery Service (URDS). The proposed architecture addresses the following issues: a) dynamic discovery of heterogeneous components, and b) selection

[1] This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

of components meeting the necessary requirements, including desired levels of QoS (Quality of Service). This paper also compares the URDS architecture with other Resource Discovery Protocols, outlining the gaps that URDS is trying to bridge.

1. Introduction

Software realizations of distributed-computing systems (DCS) are currently being based on the notions of independently created and deployed components, with public interfaces and private implementations, loosely integrating with one another to form a coalition of distributed software components. Assembling such systems requires either automatic or semi-automatic integration of software components, taking into account the quality of service (QoS) constraints advertised by each component and the collection of components. The UniFrame Approach (UA) [12][13] provides a framework that allows an interoperation of heterogeneous and distributed software components and incorporates the following key concepts: a) a meta-component model (the Unified Meta Model – UMM [11]), b) an integration of QoS at the individual component and distributed system levels, c) the validation and assurance of QoS, based on the concept of event grammars, and e) generative rules, along with their formal specifications, for assembling an ensemble of components out of available choices. The UniFrame approach depends on the discovery of independently deployed software components in a networked environment. This paper describes an architecture, URDS (UniFrame Resource Discovery Service), for the resource discovery aspect of UniFrame. The URDS architecture provides services for an automated discovery and selection of components meeting the necessary QoS requirements. URDS is designed as a Discovery Service wherein new services are dynamically discovered while providing clients with a Directory style access to services. The result of using URDS, the UA and its associated tools is a semi-automatic construction of a distributed system.

The rest of the paper is organized as follows. Section 2 discusses related resource discovery protocols. Section 3 discusses the UniFrame approach and the URDS architecture. An example is presented in section 4. A brief comparison of URDS and other approaches is presented in section 5. Details of an initial prototype and experimentations are indicated in section 6 and the paper concludes in section 7.

2. Related Work

The protocols for resource discovery can be broadly categorized into: a) ***Lookup (Directory) Services and Static Registries and b) Discovery Services.*** A few prominent approaches are briefly discussed below.

Universal Description, Discovery and Integration (UDDI) Registry: UDDI [16] specifications provide for distributed Web-based information registries wherein Web services can be published and discovered. Web Services in UDDI are described using Web Services Description Language (WSDL) [4] -- an XML grammar for describing the capabilities and technical details of Simple Object Access Protocol (SOAP) [1] based web services.

CORBA Trader Services: The CORBA Trader Service [10] facilitates 'matchmaking' between service providers (*Exporters*) and service consumers (*Importers*). The exporters register their services with the trader and the importers query the trader. The trader will find a match for the client based on the search criteria. Traders can be linked to form a *federation of traders,* thus making the offer spaces of other traders implicitly available to its own clients.

Service Location Protocol (SLP): SLP [6] architecture comprises of *User Agents (UA), Service Agents (SA),* and *Directory Agents (DA).* UAs perform service discovery on behalf of clients, SAs advertise the location and characteristics of services and DAs act as directories which aggregate service information received from SAs in their database and respond to service requests from UAs. Service requests may match according to service type or by attributes.

JINI: JINI [15] is a Java-based framework for spontaneous discovery. The main components of a JINI system are *Services*, *Clients* and *Lookup Services*. A service registers a "service proxy" with the Lookup Service and clients requesting services get a handle to the "service proxy" from the Lookup Service.

Ninja Secure Service Discovery Service (SSDS): The main components of the SSDS [5], [9] are: Service Discovery Servers (SDS), Services and Clients. SSDS shares similarities with other discovery protocols, with significant improvements in reliability, scalability, and security.

3. UniFrame and UniFrame Resource Discovery Service (URDS)

The Directory and Discovery Services, described earlier, mostly do not take advantage of the heterogeneity, local autonomy and the open architecture that are characteristics of DCS. Also, a majority of these systems operate in one-model environment (e.g., CORBA Trader service assumes only the presence of CORBA components). In contrast, a software realization of a DCS will most certainly require a combination of heterogeneous components – i.e., components developed under different models. In such a scenario, there is a need for a discovery system that exploits the open nature, heterogeneity and local autonomy inherent in DCS. The URDS architecture is one such solution for the discovery of heterogeneous and distributed software components.

3.1. UniFrame Approach

3.1.1. Components, Services and QoS

Components in UniFrame are autonomous entities, whose implementations are non-uniform. Each component has a state, an identity, a behavior, well-defined public interfaces and private implementation. In addition, each component has three aspects: a) ***Computational Aspect:*** it reflects the task(s) carried out by each component, b) ***Cooperative Aspect:*** it indicates the interaction with other components, and c) ***Auxiliary Aspect:*** this addresses other important features of a component such as security and fault tolerance.

Services, offered by a component in UniFrame, could be an intensive computational effort or an access to underlying resources. The ***QoS*** is an indication given by a software component about its confidence to carry out the required services in spite of the constantly changing execution environment and a possibility of partial failures.

3.1.2. Service Types

Components in UniFrame are specified informally in XML using a standard format. XML [3] is selected as it is general enough to express the required concepts, it is rigorously specified, and it is universally accepted and deployed. The UniFrame service type, which represents the information needed to describe a service, comprises of:

ID: A unique identifier comprising of the host name on which the component is running and the name with which this component binds itself to a registry will identify each service.

ComponentName: The name with which the service component identifies itself.

Description: A brief description of the purpose of this service component.

Function Descriptions: A brief description of each of the functions supported by the service component.

Syntactic Contracts: A definition of the computational signature of the service interface.

Function: Overall function of the service component.

Algorithm: The algorithms implemented by this component.

Complexity: The overall order of complexity of the algorithms implemented by this component.

Technology: The technology used to implement this component (e.g., CORBA, Java RMI, etc.).

QoS Metrics: Zero or more *Quality Of Service (QoS) types*. The *QoS type* defines the QoS value type. Associated with a QoS type is the triple of *<QoS-type-name, measure, value>* where *QoS-type-name* specifies the QoS metric, for example, *throughput, capacity, end-to-end delay,* etc. *Measure* indicates the quantification parameter for this type-name like *methods completed/sec, number of concurrent requests handled, time,* etc. *Value* indicates a numeric/string/boolean value for this parameter. We have established a catalog of Quality of Service metrics that are used in UniFrame specifications [2].

Figure 1 illustrates a sample UniFrame specification. This example is for a bank account management system with services for deposit, withdraw, and check balance. This example assumes the presence of a Java RMI server program and a CORBA server program, which are available to interact with the client requesting their services. We will return to this example in detail when we describe the resource discovery service.

```
<UniFrame>

    <ComponentName> AccountServer </ComponentName>
    <Description> Provides an Account Management System </Description>

    <FunctionDescription>
        <Function> javaDeposit </Function>
        <Function>javaWithdraw </Function>
        <Function> javaBalance </Function>
    </FunctionDescription>

    <ComputationalAttributes>
        <InherentAttributes>
            <ID> intrepid.cs.iupui.edu/AccountServer </ID>
        </InherentAttributes>
    </ComputationalAttributes>

    <FunctionalAttributes>
        <Function> Acts as Account Server </Function>
        <Algorithm> Simple Addition/Subtraction </Algorithm>
        <Complexity> O(1) </Complexity>
        <SyntacticContract>
            <Contract> void javaDeposit(float ip) </Contract>
            <Contract> void javaWithdraw throws OverDrawException</Contract>
            <Contract> float javaBalance() </Contract>
        </SyntacticContract>
        <Technology> Java-RMI </Technology>
    </FunctionalAttributes>

    <CooperatingAttributes>
        <PreprocessingCollaborators> AccountClient </PreprocessingCollaborators>
    </CooperatingAttributes>

    <AuxillaryAttributes>
        <Mobility> No </Mobility>
    </AuxillaryAttributes>

    <QOSMetrics>
        <Availability measure="%"> 90 </Availability>
        <End2EndDelay measure="ms" > 10 </End2EndDelay>
    </QOSMetrics>

</UniFrame>
```

Figure 1: Sample UniFrame Specification in XML

3.2 URDS

The main components of the URDS architecture (illustrated in Figure 2) are: i) Internet Component Broker (ICB), ii) Headhunters (HHs), iii) Meta-Repositories, iv) Active-Registries, v) Services, and vi) Clients. Other details in the figure will be explained in the following sections. The numbers indicate the flow of activities in the URDS. These are explained, in detail, in the context of an example in section 3.2.7. The URDS architecture is organized as a federation in order to achieve scalability. Figure 3 illustrates the federation aspect of URDS.

Every ICB has zero or more headhunters attached to it. The ICBs in turn are linked together with unidirectional links to form a directed graph. The URDS discovery process is "administratively scoped", i.e., it locates services within an administratively defined logical domain. '*Domain*' in UniFrame refers to industry specific markets such as Financial Services, Health Care Services, Manufacturing Services, etc.

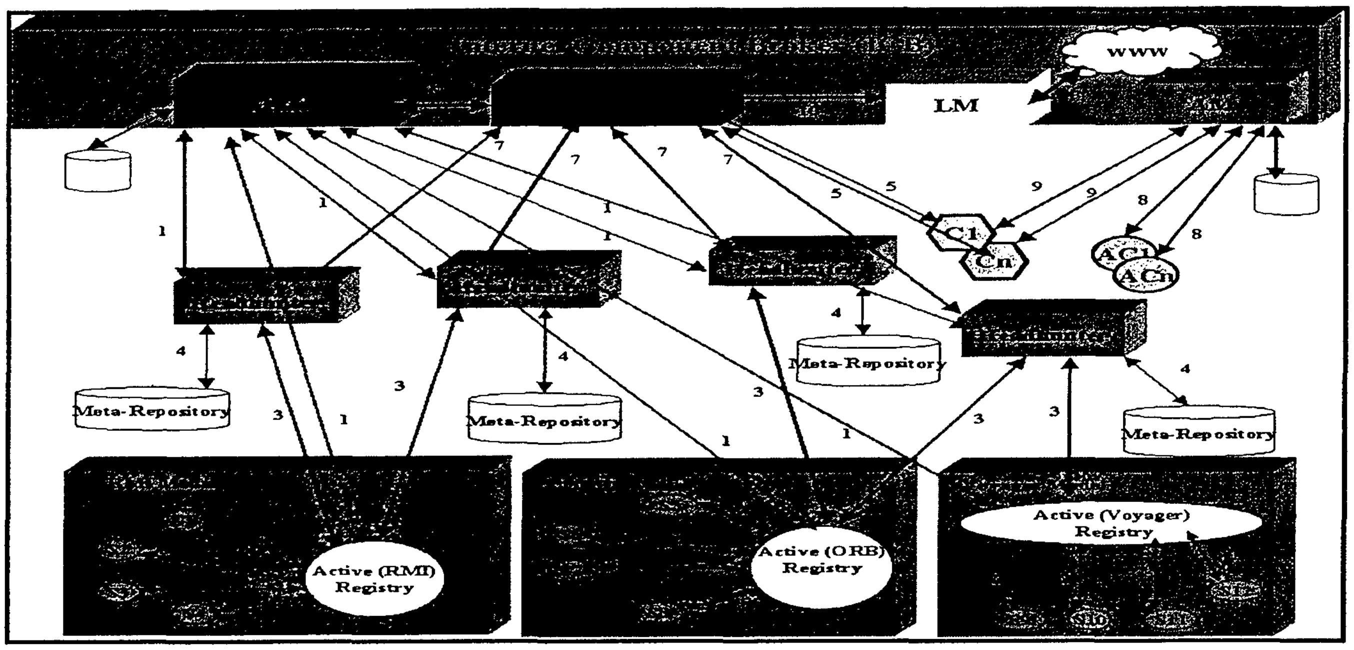


Figure 2: URDS Architecture

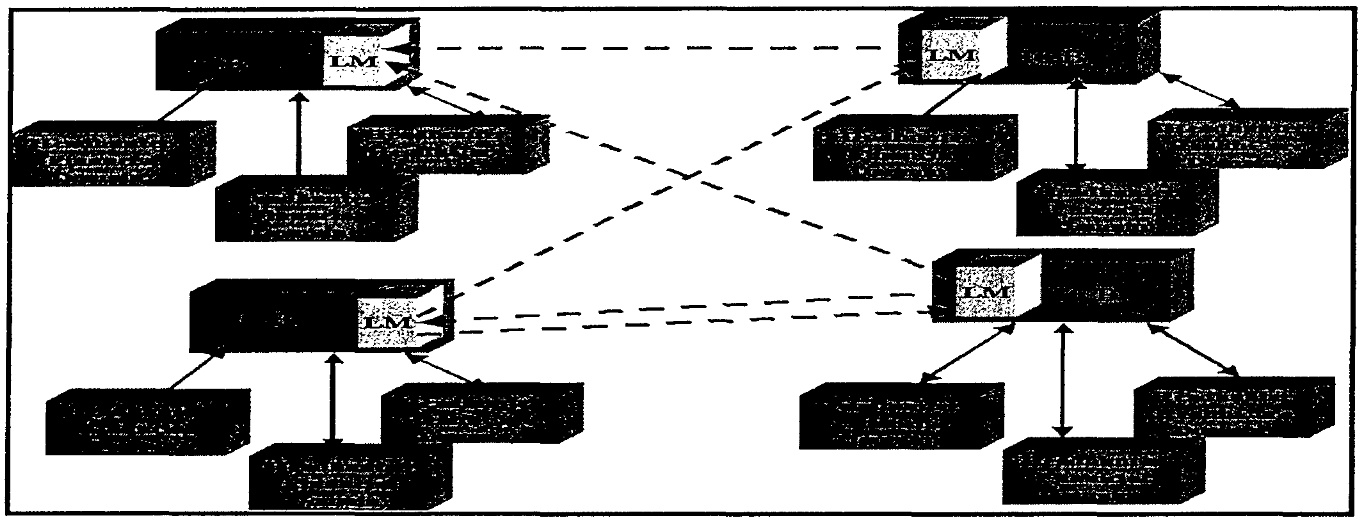


Figure 3: Federated Organization in URDS

3.2.1 Internet Component Broker (ICB)

The ICB acts as an all-pervasive component broker in the interconnected environment providing a platform for the discovery and seamless integration of disparate components. The ICB is not a single component but is a collection of services comprising of the Query Manager (QM), the Domain Security Manager (DSM), Adapter Manager (AM), and the Link Manager (LM). It is envisioned that there will be a fixed number of ICBs deployed at well-known locations hosted by corporations or organizations supporting this initiative.

The functionality of the ICB is similar to that of an Object Request Broker. However, the ICB has certain key features that are unique. It provides component mappings and component model adapters. The ICB, in conjunction with headhunters, provides the infrastructure necessary for scalable, reliable, and secure collaborative business using the interconnected infrastructure. The functionalities of the ICB are:

- Authenticate the users (Headhunters and Active Registries) in the system and enforce access control over the multicast address resources for a domain with the help of the Domain Security Manager (DSM).
- Attempt at matchmaking between service producers and consumers with the help of the Headhunters and Query Manager. ICBs may cooperate with each other in order to increase the search space for matchmaking. The cooperation techniques of ICBs are facilitated through the Link Manager (LM).
- Act as a mediator between two components adhering to different component models. The mediation capabilities of the ICB are facilitated through the Adapter Manager (AM).

Domain Security Manager (DSM)

The DSM handles secret key generation and distribution and enforces the group membership and access control to multicast resources through authentication and use of access control lists (ACL). The resources being guarded are the multicast addresses allocated to a particular *domain*. The DSM serves as an authorized third party, which maintains an inclusion list of Principals (headhunters or registries), corresponding to a domain. DSM has an associated repository (database) of valid principals, passwords, multicast address resources and domains. Every Headhunter or Active Registry is associated with a domain. The Active Registries associated with a domain have components registered with them, which belong to that domain. The Headhunter in turn detects Registries, which belong to the same domain as itself, and hence the service components detected by the headhunter will belong to a particular domain. The Principal (authenticated user), is allowed access only to the multicast address mapped to the domain with which it is associated. A Principal that wishes to participate in the discovery process contacts the DSM with its credentials (id, password, domain). The DSM authenticates the principal and checks its authorizations against the domain ACL. The DSM returns a secret key and a multicast address mapped to the corresponding domain to a valid principal. In case the principal is a Headhunter the DSM registers the contact information of the headhunter with itself. The QM to propagate queries uses this information.

Query Manager (QM)

The QM uses a natural language parser [7] to translate a service consumer's natural language-like query into an XML based query. The QM parses the XML based query to generate a structured query language statement and dispatches this query to the 'appropriate' Headhunters. The QM obtains the list of registered Headhunters from the DSM. The HH returns the list of matching service providers. The QM in conjunction with the LM is also responsible for propagating the queries to other linked ICBs. The functions performed by the QM are:

- Parse a service consumer's natural language-like query and extract the keywords and phrases pertaining to various attributes of the components UniFrame specification.
- Extract the consumer-specified constraints, preferences and policies to be applied to the various attributes.
- Compose the extracted information into an XML based query.
- Translate the XML based query to a structured query language statement.
- Dispatch this structured query to all the headhunters associated with the domain on which the search is being performed and also forward the query to the Link Manager, which will propagate the query to other ICBs.
- The headhunters will query the Meta-Repository and return a list of components matching the search criteria to the QM.
- QM will wait for a specified time period for results to be returned from the headhunters/other ICBs before timing out.
- The client has the option to specify search-scoping policies to affect the time spent on the search process.

Link Manager (LM)

ICBs are linked to form a *Federation of Brokers* (see Figure 3) in order to allow for an effective utilization of the *distributed offer space.* ICBs propagate the search query issued by the Clients to other ICBs to which they are linked apart from the headhunters with which they are associated. The LM performs the functions of the ICB associated with establishing links and propagating the queries. *Links* represent paths for propagation of queries from a source ICB to a target ICB. The LM supports the following operations:

- *Register:* LMs register with each other to create unidirectional links from the Source LM to the Target LM. The registration information comprises of the location information of the LM.
- *Query:* The query operation is responsible for propagating the query from the source LM to the list of Target LMs with which the Source LM is registered.
- *Failure Detection:* This involves keeping track of LMs that may no longer be active due to failures. Periodically each LM sends a unicast message to all other LMs that are registered with it. LMs receiving the message maintain a cache of the pairs <*Sender LM address, Time-stamp of receipt*>. At regular time intervals the receiving LMs note the 'freshness' of the information they hold and purge the Sender's information, which they deem to be 'stale'. Staleness is determined by the time elapsed between the receipt of the LM address through the unicast communication and the current time.

- *Link Traversal Control:* The Link Traversal Control mechanism used in the LM is similar to that of CORBA Trader Services. The necessity for Link Traversal Control arises due to the nature of LM linkage, which allows arbitrary, directed graphs of LMs to be produced. This can introduce two problems: i) a single LM can be visited more than once, and ii) loops can occur. To ensure that a search does not enter into an infinite loop, a **hop count** is used to limit the depth of links to propagate a search. The hop count is decremented by one before propagating a query to other LMs. The search propagation terminates at the LM when the hop count reaches zero.

Adapter Manager (AM)

The AM serves as a registry/lookup service for clients seeking adapter components. The adapter components register with the AM and while doing so they indicate their specialization (i.e., which heterogeneous component models they can bridge efficiently). Clients contact the AM to search for adapter components matching their needs. The AM utilizes adapter technology, each adapter component providing translation capabilities for specific component architectures. The adapter components achieve interoperability using the principles of wrap and glue technology [8].

3.2.2 Headhunters

Another critical component of URDS is a headhunter. The headhunters perform the following tasks: a) Service Discovery: detect the presence of service providers (Exporters), b) register the functionality of these service providers, and c) return a list of service providers to the ICB that matches the requirements of the consumer (Importers) requests forwarded by the QM.

The service discovery process utilizes a search technique based on multicasting. Once deployed in the system, the headhunters periodically multicast their presence to a multicast group. The multicast group address is obtained from the DSM. The active registries, that also obtain a multicast group address from the DSM, listen for these multicast messages. The active registries maintain a cache of the pairs <*headhunter address, time-stamp of receipt*> and periodically send response messages to all the headhunters in their cache. The headhunter in turn maintains a cache of the pairs <*registry address, time-stamp of receipt*>. The Headhunter intermittently queries the Registries for the component information of service providers they contain. During the registration, the headhunter stores into the meta-repository all the details of the service providers, including the UniFrame specifications. The headhunter uses this information during matching. A component may be registered with multiple headhunters. The functionality of headhunters makes it necessary for them to communicate with Active Registries belonging to any model, implying that the cooperative aspect of headhunters be universal. The headhunters need to also address the issues of failures and security.

- ***Failure Detection:*** Failure detection involves keeping track of service exporters that may no longer be active in the system for various reasons. Headhunters achieve failure detection at the level of detecting failures of the active registries, which hold the service exporters. The headhunter keeps track of the time at which it obtains registry location information from various active registries. At regular time intervals the headhunter notes the 'freshness' of the information it holds and purges the registry information, which it deems to be 'stale'. 'Fresh' or 'Stale' are determined based on the time elapsed between the receipt of the registry address through unicast communication and the current time. This process is based on the principle that if a registry is still active in the system, it will respond to the headhunter with its location information and thus have a recent timestamp. A registry which for whatever reason is unable to contact the headhunter with its information will hold a 'stale' timestamp and it will be assumed that all service exporter components held by this registry are no longer available for rendering their services.

- ***Multicast Security:*** This involves securing the multicast data transmission mechanism from security threats such as eavesdropping, and masquerading. The headhunter uses Secret Key Encryption to ensure security of transmitted data. The secret key used is a symmetric key wherein the sender and receiver use the same key for purposes of encryption and decryption.

3.2.3 Meta-Repository

The Meta-Repository is a data store that serves to hold service information of exporters adhering to different models. The service information stored by the Meta-repository consists of: a) Service type name, b) Details of its informal specification, and c) Zero or more QoS values for that service for each of the components. The implementation of a Meta-Repository is database oriented. A Meta-Repository is a *passive component,* i.e., a headhunter brings information to the meta-repository.

3.2.4 Active Registry

The native registries (e.g., RMI Registry or CORBA registry) are extended to have the following features:

- *Activeness:* The registries are modified to be able to listen to multicast messages from the headhunter and respond with their host IP Address.

- *Introspection Capabilities:* The registries are extended to not only keep a list of component URLs of those components registered with them but also their detailed UniFrame specifications. This is achieved by querying the components (using principles of introspection) to obtain the URL of their XML based specifications. The registries parse the specification and maintain the details in a memory resident table, which is returned to the headhunter upon request.

- *Failure Detection Of Headhunters:* Failure detection involves keeping track of headhunters, which may no longer be active in the system for reasons such as network or node failure. The active registries keep track of the time at which it obtains headhunter location information from various headhunters through multicast. At regular intervals the active registries note the 'freshness' of the headhunter information they hold and purge the headhunter information, which they deem to be 'stale'. 'Fresh' or 'stale' are determined based on the time elapsed between the receipt of the headhunter address through multicast communication and the current time.

3.2.5 Service Exporter Components

Service Exporter Components are implemented in different models, e.g., Java RMI, CORBA, EJB, etc. The components are identified by their *Service Offers* comprising of service type name, b) informal UniFrame specification, and c) zero or more QoS values for that service. The component registers its interfaces with its local registry. The component interface contains a method, which returns the URL of its informal specification. The informal specification is stored as a XML file adhering to certain syntactic contracts to facilitate parsing. These service exporter components will be tailored for specific domains, such as Financial Services, and will adhere to the relevant standards in those domains.

3.2.6 Clients

Clients are Service Requesters searching for services matching certain functional and non-functional requirements.

4. An Example

Table 1 outlines the interactions between the URDS components in servicing a client query for assembling an account management system. The rows of the table are numbered corresponding to the flow of control shown in Figure 2. The result of this interaction will be an ensemble of components, which may be assembled into a complete system as described in [12].

Table 1: Interactions between URDS components

1	This indicates the interactions between the principals (Headhunters/Active registries) and the DSM.
	• The principals contact the DSM with their authentication credentials in order to obtain the secret key and multicast address for group communication (many to one interaction).
	`<name="headhunter1", password="secret1", domain="financial">`
	`<name="registry2", password="secret2", domain="financial">`
	• The DSM authenticates the principals and returns a secret key and multicast address to a valid principal (one to many interaction).
	`<secretkey = key.dat, multicast_address="224.2.2.2">`

2	This indicates the interactions between Service Exporter Components and active registries. • Service exporter components register with their respective registries (many to one interaction) -- `<id="intrepid.cs.iupui.edu/AccountServer">`

	• These registries in turn query these components for their UniFrame Specification (one to many interaction).
	`<introspect property = "uniFrameSpecURL">`
	• The components respond with the URL at which the specification is located (any to one interaction).
	`<url="C:\Account System\AccountServerSpec.xml">`

3	This indicates the interactions between Headhunters and Active Registries.
	• Headhunters periodically multicast their presence to a multicast group addresses (one to many interaction).
	`<headhunterlocation = phoenix.cs.iupui.edu/headhunter1>`
	• Active Registries, which are listening at this group address, respond to Headhunters' messages by passing their information to Headhunters (many to many interaction).
	`<registrylocation = magellan.cs.iupui.edu/registry2>`
	• Headhunters intermittently query the active registries to which they hold a reference for the information of all the components registered with them (one to many interaction). The active registries respond by passing the list of components registered with them and the detailed UniFrame specification of these components (many to many interaction).

4	This indicates the interactions between a Headhunter and a Meta-Repository. • Headhunters persist the component information obtain ed from the active registries onto the Meta-Repository (one to one interaction). • Headhunters query Meta-Repository to retrieve component information (one to one interaction). `<query="SELECT * FROM` *`componentTable`* `A,` *`functionTable`* `B WHERE (A.ID = B.ID) AND ((`*`description`* `LIKE%account%) OR (`*`description`* `LIKE %system%)) AND (`*`end2endDelay`*`<10) AND (`*`availability`*`>90)">` • Meta-Repository returns search results to headhunter (one to one interaction).
5	This indicates the interactions between the QM and clients. • Clients contact the QM and specify the functional and non-functional search criteria (many to one interaction).

	• The natural language-like client query is as follows: “Create an account management system that has end-to-end delay < 10 ms and availability > 90% preference maximum availability”. • Figure 4 shows the translated XML based query. `<Query>` `<Description> Account System </Description>` `<Domain> Financial </Domain>` `<End2EndDelay constraint="<">10 </End2EndDelay>` `<Availability constraint = “>” preference ="max"> 90 </Availability>` `</Query>` Figure 4: Processed XML query • The QM returns the search results to the clients (one to many interaction). `< component 1: id="..", description="…", availability="…",…;` `component 2: id="..", description="…", availability="…"…;` `component 3: id="..", description="…", availability="…",…;>`
6	This indicates the interaction between the QM and DSM. • QM contacts DSM for contact information of registered headhunters belonging to the domain of client query (one to one interaction). • DSM responds with list of registered headhunters (one to one interaction). `<phoenix.cs.iupui.edu/headhunter1` `magellan.cs.iupui.edu/headhunter2>`
7	This indicates the interactions between the QM and headhunters. • The QM propagates Client’s query to all headhunters registered with it, which fall in the domain of the Client’s search request (one to many interaction). • The headhunters respond to the QM query with search results matching the criteria (many to one interaction).
8	This indicates the interactions between adapter components and AM. • Adapter components register with the AM, which is running at a well-known location (many to one interaction).

9	This shows the interactions between the clients and the AM. • Clients contact the AM at the well-known location at which it is running with requests for specific adapter components (many to one interaction). • The AM checks against its repository for matches and returns the results to the clients (one to many interaction).
10	This shows the interactions between QM and LM. • The QM propagates the query to the LM (one to one interaction). • LM returns search results to QM (one to one interaction).
11	This shows the interactions between the LM of one ICB and target LMs of other ICBs with which this LM is registered. • The LM propagates the search query issued by the QM to the target LMs (one to many interaction). • The source LM receives the result responses from these target LMs (many to one interaction).

5. Comparison between URDS and Other Resource Discovery Protocols

A brief comparison between URDS and other approaches is provided below.

- *Interoperability:* The other resource discovery protocols provide services for specific models and interoperations can be achieved only through proxies. URDS addresses the issue of non-uniformity by providing for discovery and coordination between components implemented using diverse models.
- *Network Usage:* Unlike other protocols, URDS clients and services do not participate in active discovery thus cutting down on the periodic communication required for the process of discovery. Instead, the active nature of the extended native registries allows the discovery process and removes the additional burden of developing 'active' components.
- *Query Processing and Matchmaking:* Unlike other approaches, which rely on Java-based or XML-based matching, the URDS supports a natural language-like query mechanism. This provides flexibility in formatting queries and during the matchmaking process.
- *Domain of Discovery:* In URDS the contextualization of the search space is logical and dictated by the industry specific markets. In other discovery protocols the

notion of "administrative scope" is associated with the topology of the network domain.

- *Security:* The URDS security model addresses many of the common threats, which may occur during the discovery process. SSDS is another service notable for its robust security model.
- *QoS:* UniFrame incorporates of the notion of QoS as applied to software components and integrates this aspect into the service specification and the matchmaking process.

6. Prototype and Experimentation

A preliminary prototype [14] of the URDS has been implemented using the J2EE version 1.4 software environment. The core architectural components (domain security manager, query manager, link manager, headhunters and active registries) have been implemented as Java-RMI based services.

The repositories (domain security manager's repository and meta repository) have been implemented using Oracle version 8.0. The Web-based components (JSPs), which service client interactions, are placed in a Tomcat 4.0 Servlet/JSP container.

The unicast communication between the core architectural components is achieved through JRMP (Java Remote Method Protocol) and the multicast communication between the headhunters and the active registries is achieved through Multicast sockets based on UDP/IP. The database connections are established using the JDBC (Java Database Connectivity) APIs and the user interaction is achieved through a browser front-end using the HTTP protocol. The security infrastructure, of URDS, is implemented by the security and cryptography APIs that form a part of Java Cryptography Architecture and Java Cryptographic Extension frameworks.

Preliminary experiments were carried out on this prototype to observe the performance of URDS. The experimental setup consisted of Sun SPARC machines connected by an Ethernet. The experiments contained one ICB, one headhunter, and one active registry (enhanced version of Java RMI registry). A single client was used to issue query requests, which consisted of different QoS constraints. The measurements were averaged over one hundred trials. The following times were measured:

- Average Authentication Time: It is the average time taken by the domain security manager to authenticate a principal (i.e., headhunter and active registry).
- Average Query Service Time: It is the average time taken to service a query.
- Average Registry Discovery Time: It is the average time taken by a headhunter to discover an active registry.
- Average Component Information Retrieval Time: It is the average time taken by the headhunter to retrieve component information from an active registry.

These initial experiments showed a value of 690 ms for the average authentication time. The average query time and the registry discovery time showed a marginal increase with an increasing number of registered components; while the average component retrieval information time increased linearly with the number of components (as expected).

The current prototype is able to discover only Java-RMI components, thus making it homogeneous. Efforts are underway to make it heterogeneous, i.e., able to discover components created using other models (such as CORBA, .NET, etc.) also. The current prototype also does not include the federation aspect.

7. Conclusion

The paper has presented an architecture that facilitates the semi-automatic construction of a distributed system by providing for the dynamic discovery of heterogeneous components and selection of components meeting the necessary requirements, including desired levels of QoS. The URDS architecture addresses issues such as interoperability, QoS of software components, scalability, fault tolerance, security and network usage. Interoperability is achieved by discovering components developed in several different component models. The discovery mechanism uses multicasting to detect native registries/lookup services of various component models that are extended to possess 'active' and 'introspective' capabilities. The component specification captures their computational, functional, co-operational, auxiliary attributes and QoS metrics. Flexibility in query formatting is achieved by providing support for natural language-like client requests. As a scalability mechanism URDS is organized in a federated hierarchical structure. Failure tolerance is handled through periodic announcements by entities and through information caching. Security is provided through authentication of the principals involved, access control to multicast address resources, and encryption of data transmitted. URDS provides a directory based discovery service which is scalable secure and fault tolerant. Although, the current prototype does not address all the features of the URDS architecture, it has created a basis for validating the concepts behind URDS. Efforts are underway to extend the current prototype that will enable a validation of all the features presented in this paper.

References

[1] Box, D., et al., "Simple Object Access Protocol (SOAP) 1.1", W3C, May 2000, http://www.w3.org/TR/SOAP.

[2] Brahmnath, G., Raje, R. R., Olson, A. M., Auguston, M., Bryant, B. R., Burt, C. C., "A Quality of Service Catalog for Software Components," to appear in Proceedings of the 2002 Southeastern Software Engineering Conference, 2002.

[3] Bray, T., Paoli, J., Sperberg-McQueen, C. M. "Extensible Markup Language (XML) 1.0 (Second Edition)," W3C, October 2000, http: //www.w3c.org/xml.

[4] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., "Web Services Description Language (WSDL) 1.1," W3C, March 2001 http://www.w3.org/TR/wsdl.

[5] Czerwinski, S. E., Zhao, B. Y., Hodes, T. D., Joseph, A. D., Katz, R. H., "An Architecture for a Secure Service Discovery Service," Proceedings of Mobicom '99, 1999. http://ninja.cs.berkeley.edu/dist/papers/sds-mobicom.pdf

[6] Guttman, E., "Service Location Protocol: Automatic Discovery of IP Network Services," IEEE Internet Computing, vol. 3, no. 4, 1999, pp. 71-80.

[7] Lee, B.-S., and Bryant, Barrett R., "Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language," Proceedings of SAC 2002, the ACM Symposium on Applied Computing, 2002, pp. 932-936.

[8] Luqi, Berzins, V., Ge, J., Shing, M., Auguston, M., Bryant, B. R., Kin, B. K., "DCAPS – Architecture for Distributed Computer Aided Prototyping System," Proceedings of RSP 2001, the 12th Rapid Systems Prototyping Workshop, 2001, pp. 103-108.

[9] Ninja, "The Ninja Project," http://ninja.cs.berkeley.edu, 2002.

[10] Object Management Group, "Trading Object Service Specification," Object Management Group 2000. ftp://ftp.omg.org/pub/docs/formal/00-06-27.pdf.

[11] Raje, R. R., "UMM: Unified Meta-object Model for Open Distributed Systems", Proceedings of ICA3PP 2000, 4th IEEE Int. Conf. Algorithms and Architecture for Parallel Processing", 2000, pp. 454-465.

[12] Raje, R., Auguston, M., Bryant, B. R., Olson, A., Burt, C., "A Unified Approach for the Integration of Distributed Heterogeneous Software Components", Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration, 2001, pp. 109-119.

[13] Raje, R., Auguston, M., Bryant, B. R., Olson, A., Burt, C., "A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components", Technical Report, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2002.

[14] Siram, N. N., "An Architecture for the UniFrame Resource Discovery Service", MS Thesis, Indiana University Purdue University Indianapolis, Spring 2002.

[15] Sun Microsystems, "Jini Architecture Specification, Version 1.2," Sun Microsystems, December 2001, http://www.sun.com/jini/.

[16] uddi.org, "UDDI Technical White Paper", September 2000, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models*

Carol C. Burt
Barrett R. Bryant
University of Alabama Birmingham
cburt, bryant@cis.uab.edu

Rajeev R. Raje
Andrew Olson
Indiana University Purdue University Indianapolis
rraje,aolson@cs.iupui.edu

Mikhail Auguston
New Mexico State University
mikau@cs.nmsu.edu

Abstract

The UniFrame research project is proposing a Unified Component Meta Model Framework (UniFrame) that includes Quality of Service (QoS) contracts. Today it is the role of the software architect, based on experience, to design platform specific solutions that will meet QoS requirements. As we refine algorithms for model transformations, we must identify these QoS-aware design patterns and utilize them during model transformations. Our research includes supporting and participating in the exploration of generative techniques as they relate to QoS requirements (both static and dynamic) and the standardization of QoS-aware transformations. This paper explores how QoS requirements can impact decisions related to model transformation (using UML for Platform Independent Modeling and ISO IDL for the Platform Specific Model). It explores a series of QoS related design issues that must be considered as platform independent models are refined for specific component platforms.

1. Introduction

Enterprises are increasingly dependent upon multiple middleware technologies that enable new business paradigms by weaving together legacy systems with advanced technology. This technology supports core business functionality, enables distributed business systems, integrates business processes and enables companies to communicate with customers, suppliers, and business partners. While it is possible to construct heterogeneous component systems, it requires that the developer be aware of the nuances of the diverse middleware technologies. In addition, the increased complexity of this environment makes it impossible to predict the non-functional aspects of such a system until after it is constructed. That is, metrics and test scenarios must be hand crafted on a case-by-case basis to determine if a composition is acceptable. These problems must be resolved for the promise of software component technology to be fully realized.

The Unified Component Meta-Model Framework (UniFrame) [1] research is an attempt to unify distributed component models under a common meta-model for the purpose of enabling the discovery, interoperability, and collaboration of components via generative software techniques. This research targets the dynamic assembly of distributed software systems from components under different component models, and explores how the quality of service (QoS) requirements influences the design of components and their compositions.

Today, software architects leverage their experience in designing distributed systems when refining business and information technology models to ensure the quality of service requirements are met. To enable the use of generative techniques as models are refined, these experience-based design patterns must be formalized. As a part of this research, we plan to document the effect of design decisions on attaining quality of service requirements and explore techniques for providing the instrumentation necessary to measure QoS features. In this research we are focusing on two key QoS aspects for distributed component solutions: the security access control and the performance.

This paper explores the experience-based design considerations related to quality of service requirements during the model transformations when the Model Driven Architecture [2] techniques are used. It expands on previous work [3] that identified standards that are in progress as well as additional standards that are needed for the definition of QoS-based service contracts. For illustrative purposes, it presents design considerations

* This research was supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

for the security and the performance during the transformation of a simple Platform Independent Model (described in UML) to a CORBA model (described in ISO IDL). The future goals of our research include the identification and standardization of metrics necessary to validate the patterns and a mechanism to allow QoS related design patterns to be expressed as model parameters.

2. Model Driven Architecture

Model driven architecture techniques are not new; business and process modeling have been used for many years to capture requirements of information systems. As object-oriented analysis and design techniques matured, the Unified Modeling Language (UML) was standardized by OMG and became a popular technique for expressing both domain/business models and models of information systems.

OMG's Model Driven Architecture (MDA) [2] initiative facilitates the standardization of Platform Independent Model (PIMs) and the transformation of those models to multiple Platform Specific Models for implementation (such as CORBA, J2EE, or Web Services). In this way a single PIM can be used as the basis for multiple implementation technologies, and with standardization of the transformation algorithms, appropriate bridges can be generated. Standardizing platform independent models is a natural extension of existing OMG analysis and design standards for modeling and meta-modeling services. Standardizing multiple transformations to diverse technology platforms is a natural extension of the OMG mission to define interoperability standards.

Many OMG standards contain UML models to describe the domain model and/or semantics of services. Typically these domain models (expressed or implied) are independent of the CORBA platform (evidenced by the fact that they have been leveraged for use in J2EE and other technology platforms). In the past, OMG has only standardized the transformations to CORBA specific model(s) expressed in ISO IDL; however, it is expected that many of the existing services will be standardized for alternative platform technologies.

This focus on the Model Driven Architecture is a catalyst for the consideration of the effects of Quality of Service (QoS) requirements on computing models. At present, we have a limited ability to express QoS requirements as model parameters and even less definition of the algorithmic requirements to satisfy specific quality of service demands. The Model Driven Architectural vision, which is consistent with those of this research, includes standards that enable the use of generative techniques for construction of interoperability bridges between platform technologies. While this vision is appealing, there is a great deal of research to be done before this is feasible. The problem lies not in determining a single transformation from a platform independent model to a platform specific model, but in understanding the appropriate transformation based on quality of service requirements. Some of the model transformation issues related to the performance and security access control are discussed in this paper.

3. Relevant Standards and Known Issues

OMG has standardized technologies [18] that include a UML profile for CORBA and a UML profile for Enterprise Distributed Object Computing (EDOC). In addition, the Java Community Process has standardized a UML profile for Java2 Enterprise Edition (J2EE). These profiles, however, do not consider how to model QoS related aspects.

The OMG Meta-Object Facility provides a standard for generation of interfaces from MOF compliant UML models. However, it is well known that there are issues with this mapping for distributed solutions. The OMG Architecture board produced a paper that describes the technical details of the Model Driven Architecture (MDA) [3]. This document outlines areas where research is required before the MDA vision can be fully realized. The paper states: "It is generally agreed that the MOF-IDL mapping is in need of upgrading. The problem is that the generated interfaces are not efficient in distributed systems. Firstly, the mapping predates CORBA valuetypes and thus does not make use of them. Secondly, a class with N attributes is always mapped to a CORBA interface with N separate getter/setter operations. In a distributed system one would want to group attributes based upon use cases, cache attribute values, or implement other optimizations to reduce the number of distributed calls. Realistically we will probably have to accept the fact that for the foreseeable future, the automatically generated transformation from PIM to PSM will have to be enhanced by humans. As we gain more experience we will be able to define various patterns and allow them to be selected in some way."

In addition, security requirements often influence the technique utilized in transformation of a platform independent model to a platform specific model. It is widely accepted within the Model Driven Architecture community that generated interfaces must be optimized using the quality of service and usage scenarios. This requires research on the appropriate techniques for integrating QoS into the generative programming model [4] is necessary before standards can be progressed in this area.

4. MDA and Quality of Service

Although QoS parameters and associated metrics have been widely used in networking, there is no standard vocabulary for discussing the QoS as it relates to distributed computing and component-based solutions. For example, the CORBA® Components Specification only uses the term "quality of service" with regard to events and whether or not they are transactional in nature [5]. The Java2 Enterprise Edition (J2EE) specification [6] clearly states the expectation that J2EE products will vary widely and compete vigorously on various aspects of quality of service. Such products will provide different levels of performance, scalability, robustness, availability, and security, although in some cases the specification requires minimal levels of service.

A standard vocabulary is the first step toward progressing Model Driven Architectures that include QoS parameterization and/or QoS contracts. This is one of the goals of the UniFrame research.

4.1 Previous and Related Work

As a part of the UniFrame research, we have outlined an approach to a QoS-based framework for creating distributed heterogeneous software components [7]. The QoS-based method in UniFrame is made up of three steps:

1. The creation of a catalog for QoS parameters (or metrics),
2. A formal specification of these parameters, and
3. A mechanism for ensuring these parameters, both at each individual component level and at the entire system level.

Our work leverages the research work by Zinky, Bakken and Schantz [8] with a goal of providing a catalog of QoS parameters and indicating how parameters might be described. There are many possible QoS parameters that a component (and its developer) can use to indicate the associated service. Some of these parameters may be general in nature, while others may pertain to a specific domain. The goal of creating the QoS catalog is two fold: a) it assists the component developer (or the system integrator) in selecting the necessary QoS parameters for the component (or system) under construction, and b) it enables the developer (or integrator) to ensure the necessary QoS guarantees by integrating the selected QoS parameters into the assurance process. We have created a preliminary version of the QoS catalog in [15]. In addition to identifying and describing different QoS parameters, this catalog also classifies them and provides models for their compositions.

Other relevant research work in this area includes Frolund and Koistinen [9] who point out that deciding which quality of service properties should be provided by individual components is an important part of the design process. They define a Quality-of-Service specification language (QML) and they show how the Unified Modeling Language (UML) can be extended to support the concepts of QML. They also show how to represent QML constructs in terms of ISO Interface Definition Language (IDL) [9] [10]. There are also case studies where Object Constraint Language (OCL) is being used a mechanism for the annotation of UML models for the purpose of expressing security constraints [11]. Recent work in adaptive systems extends the work in Quality Objects (QuO) [12] with security specific strategies that use the QuO contract definition language (QDL) [13].

We expect standards activity in this area will consider and leverage the experience and results of these efforts.

4.2 Recent Standards Activity

In January 2002, the OMG Analysis and Design task force issued a RFP (Request for Proposals) for a "UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" [14]. This RFP solicits proposals for a UML profile or Meta Object Facility (MOF) meta-model that defines standard paradigms of use in modeling quality of service and fault-tolerance aspects of systems. This is the first of a series of RFPs that have the goal of significant benefits to the UML user community engaged in high-quality robust system development. The mandatory requirements of this RFP are listed in Figure 1.

As distributed systems are becoming more omni-present with many of them handling mission-critical applications, the notion of QoS-oriented software development is of paramount importance. Such a quality-oriented approach, in addition to providing seamless access to heterogeneous components, will also ensure the reliability and a high confidence of

1. A General Quality of Service Framework

To ensure consistency in modeling various qualities of service, submissions shall define a standard framework or, reference model, for QoS modeling in the context of the UML. This shall include:

- A general categorization of different kinds of QoS; including QoS that are fixed at design time as well as ones that are managed dynamically
- Integration of different categories of QoS for the purpose of QoS modeling of system aspects.
- A coherent set of stereotypes, tagged values, and constraints as necessary to represent the identified QoS properties constructing a UML Profile.
- Identification of the basic conceptual elements involved in QoS and their mutual relationships. This shall include the ability to associate QoS characteristics to model elements (specification), a generic model of the system aspects involved in QoS-associated collaboration and their functional interactions and use cases (usage model), and a generic model of how QoS allocation and decomposition is managed.

2. A Definition of Individual QoS Characteristics

Submissions shall define QoS characteristics, particularly those important to real-time and high confidence systems, which describe the fundamental aspects of the various specific kinds of QoS based on the QoS categorization identified in the framework. These shall include but are not limited to the following:

- time-related characteristics (delays, freshness)
- importance-related characteristics (priority, precedence)
- capacity-related characteristics (throughput, capacity)
- integrity related characteristics (accuracy)
- fault tolerance characteristics (mean-time between failures, mean-time to repair, number of replicas)

3. A coherent set of stereotypes, tagged values, and constraints as necessary to represent the identified QoS properties constructing a UML Profile.

Figure 1: OMG RFP UML Profile for QoS Mandatory Requirements

distributed systems software. As indicated earlier, the need for standardization of a quality of service vocabulary was recognized early in our research and we are carefully tracking the work of the OMG in this area as we continue to progress our work in the development of a quality of service catalog [15].

5. Models Transformations

UML is a graphical notation for expressing models; it is important to understand that many alternative modeling syntax exist – for example, the XML Model Interchange (XMI) format leverages Extended Mark-up Language (XML) to express Meta-Object Facility (MOF) compliant models. While there is a standard UML profile for CORBA, the ISO IDL continues to be the most common notation used to define a CORBA model. Our research is also exploring the use of two-level grammar (TLG) as a formal mechanism for expressing models [16]. These text notations are useful for computers as they process textural or binary syntax more efficiently than graphics. Mappings from one notation to another are often produced and used for various analysis tasks (sometimes preserving all model information, and sometimes losing information which has no equivalent in an alternative modeling syntax). For example, IDL models may be expressed in the UML profile for CORBA. Such mappings are not transformations – they are merely alternative representations of the same model.

A model transformation occurs when models are refined and details are added for the purpose of focusing on a particular implementation technology or an aspect of the domain model. Model transformations are used to document different "levels of abstractions", "viewpoints" or "aspects" of an information system. Models that comply to a specific meta-model may utilize generative techniques for the transformations; leveraging information that the generator knows regarding the target implementation platform and/or parameterizations provided by the software architect. To fully realize the potential of the MDA, the Quality of Service (QoS) catalogs, the formal parameterization of Platform Independent Models, and ultimately the instrumentation generation rules must be standardized within the Model Driven Architecture roadmap.

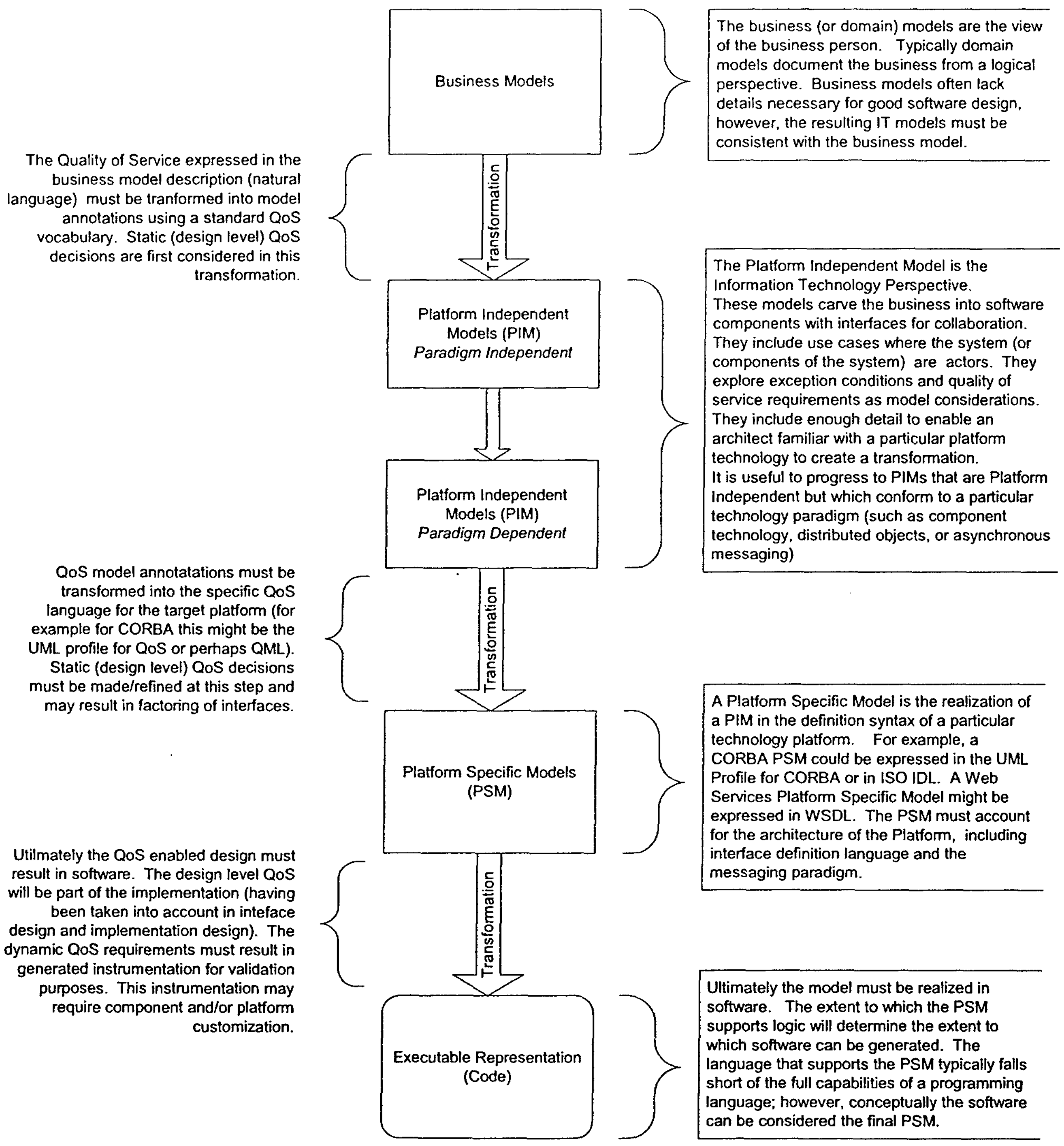

Business Models
The business (or domain) models are the view of the business person. Typically domain models document the business from a logical perspective. Business models often lack details necessary for good software design, however, the resulting IT models must be consistent with the business model.
The Quality of Service expressed in the business model description (natural language) must be tranformed into model annotations using a standard QoS vocabulary. Static (design level) QoS decisions are first considered in this transformation.
Transformation
Platform Independent Models (PIM) *Paradigm Independent*
The Platform Independent Model is the Information Technology Perspective. These models carve the business into software components with interfaces for collaboration. They include use cases where the system (or components of the system) are actors. They explore exception conditions and quality of service requirements as model considerations. They include enough detail to enable an architect familiar with a particular platform technology to create a transformation. It is useful to progress to PIMs that are Platform Independent but which conform to a particular technology paradigm (such as component technology, distributed objects, or asynchronous messaging)
Platform Independent Models (PIM) *Paradigm Dependent*
QoS model annotatations must be transformed into the specific QoS language for the target platform (for example for CORBA this might be the UML profile for QoS or perhaps QML). Static (design level) QoS decisions must be made/refined at this step and may result in factoring of interfaces.
Transformation
Platform Specific Models (PSM)
A Platform Specific Model is the realization of a PIM in the definition syntax of a particular technology platform. For example, a CORBA PSM could be expressed in the UML Profile for CORBA or in ISO IDL. A Web Services Platform Specific Model might be expressed in WSDL. The PSM must account for the architecture of the Platform, including interface definition language and the messaging paradigm.
Utilmately the QoS enabled design must result in software. The design level QoS will be part of the implementation (having been taken into account in inteface design and implementation design). The dynamic QoS requirements must result in generated instrumentation for validation purposes. This instrumentation may require component and/or platform customization.
Transformation
Executable Representation (Code)
Ultimately the model must be realized in software. The extent to which the PSM supports logic will determine the extent to which software can be generated. The language that supports the PSM typically falls short of the full capabilities of a programming language; however, conceptually the software can be considered the final PSM.

Figure 2 – QoS considerations during model transformation

Figure 2 outlines the models that are commonly progressed in a MDA approach. Quality of Service annotations or parameters must be introduced into each model and the transformations must consider such parameters as models are refined. The current OMG RFP is a beginning – standardizing a vocabulary and syntax for expressing QoS in UML. As we move beyond the QoS catalog, our research will focus on the constraints that are placed on transformations as a result of the quality requirements and explore generative techniques for ensuring that metrics can be gathered. In addition, use case scenarios must be formally expressed so that they can be used as an input to an interface generator. Thus, an ultimate goal is that given a parameterized domain model, semantically equivalent interfaces (and the bridges between them) might be generated. Our future work will explore mechanisms for expressing such parameters as annotations for design patterns so that this vision can be progressed.

In the example described below, we will follow the progression of a business model for a simple bank to a CORBA Platform Specific Model that uses experienced-based design patterns to address Quality of Service requirements. We will look at how these patterns allow security administration to be simplified and the most common remote services to be optimized. The final set of interfaces will be presented as the "UniFrameBank".

6. An Example: Model Driven Architecture with Quality of Service Considerations

Model Driven Architecture starts with the construction of a business (or domain) model based on the requirements analysis. Requirements are often expressed in a natural language and UML is a popular tool for documenting and validating the business model. In this example, we will analyze a "simple bank" and explore how interfaces may be organized based on the quality of service aspects and known use case scenarios.

6.1 Simple Bank Business Model

A typical business description of a simple bank is: The SimpleBank manages accounts. A unique account number identifies each account. An account has items associated with it. An item is a transaction against the account (deposit, withdrawal or adjustment). Deposits and withdrawals have a unique identifier, a date and an amount. Adjustments have these attributes and an annotation that provides the reason for the adjustment. There is a bank identifier or bank routing number that is used as an account prefix when interfacing with other banks. This bank id is not, however, used internally as part of the account number. Accounts maintain an owner identifier, a single PIN number, and an available balance. The SimpleBank supports the opening and closing of accounts and update of account information such as owner and PIN. Accounts are typically located using the account number, but can also be located using the owner identifier. Some business services require that the PIN be validated before the transaction can be completed.

Figure 3, based on the above description, indicates the UML business model for the simple bank.

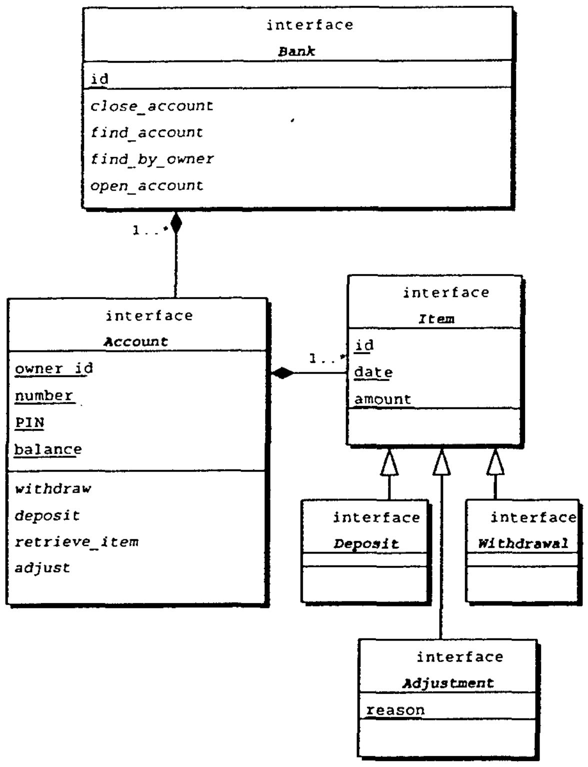


Figure 3: The simple bank business model

6.2 Simple Bank Platform Independent Model

The next step is to determine the usage scenarios that must be supported by our SimpleBank, to fully explore the business rules and to determine the quality of service characteristics of the usage scenarios (or services). This is necessary to create a Platform Independent Information Technology (IT) model of the SimpleBank that enables efficient information technology services to be offered by the SimpleBank. We need to resolve questions that arise during the development of the business model such as:

- Can one owner have multiple accounts?
- If one owner can have multiple accounts, how do we navigate to them?
- Is there a need to iterate through account items?
- What are the most common usage scenarios?
- How do we optimize the services to accommodate the common usage patterns?

A use case analysis is employed to capture this information. The Platform Independent Model considers additional details such as exceptions and security considerations that are not unique to a particular platform. Abstracting away such details is typical of business models, but those issues must be considered for an information technology system. A common initial approach to defining the Platform Independent Model is to add design details directly to the business model. This is typically not sufficient as business models are often not appropriate for expressing information technology viewpoints. For this reason, a software architect, drawing on their own experienced-based design patterns and taking all aspects of the model into consideration, transforms the business model into a PIM. This paper discusses the transformation of model and outlines some of these experienced-based techniques. It is hoped that these experience-based techniques will be formalized in the future for the purpose of using them with generative algorithms.

During use case analysis for the SimpleBank, we capture the following business rules that must be supported by the information system (this is a subset provided to aid in illustration of the QoS requirements).

- Bank customers may query account balance (via phone) and/or withdraw funds (using a teller machine) from an account without assistance provided that they have their account number and PIN.
- Merchants may request withdrawals from accounts by providing their merchant identification, account number and PIN (checkcard services).
- Tellers may locate accounts based on owner identification, query account balances, process deposits and withdrawals for customer and review existing account items. Tellers may use external means of identifying a customer (not required to use/know PIN).
- Bank managers may perform all the functions of a Teller and may also open and close accounts and create adjustments.
- Bank customers may have many accounts and will use the same owner identifier for all these accounts. It must be easy to locate all the accounts for a customer.
- The bank offers a response time guarantee of three second to merchants for services or the fees are waived for the request. Merchant requests must be prioritized above other system requests. Response times for merchant requests must be monitored.
- Account balance inquires from remote locations are a very common business scenario that requires less than five second response time to ensure customer satisfaction. Response time on balance inquires must be monitored.

The first quality of service issue we will address is one aspect of security: access control. We will use the techniques outlined in Figure 4 to review the model and use cases and apply experience-based security access control design patterns.

Are there significant security requirements identified for the service(s)?

If so, consider segregating administrative features into separate interfaces from those that provide the less restrictive non-administrative functionality

Is it expected that administrators will also be allowed to use all the non-administrative features of a service?

Use inheritance to clarify this in the model and simplify the security model. That is, an administrative interface should inherit from the non-administrative interface.

Can you navigate between interfaces as required while maintaining security controls at the point of navigation?

Review navigation patterns to ensure that given an object reference, it will be easy to navigate to other objects and that security rules logically apply at the point of navigation.

Figure 4: Experience-based Security Techniques

It is much easier for security administrators to assign policies based on roles to groups of functionality (vs. individual users and individual functions). If functionality can be grouped based on security patterns (such as view access vs. administration access) then security policy can be defined based on functional groupings (ultimately interfaces and/or objects). This also increases the scalability of the security model and is more efficient at run-time.

Our analysis review indicates that there are significant security related usage restrictions, and that using the business model as the basis of the PIM without refinement for security considerations would force access control checks for each individual operation. For example, our business rules state that the open_account() and close_account() operations can only be done by a bank manager, but they are in the same interface as the find_account() operation that locates accounts and must be accessible to tellers. In addition, we see that bank managers are allowed to perform all the functions of tellers, so we can use inheritance to capture this aspect of the access requirements. Finally, we need to review the navigation patterns to ensure that our Platform Independent Model supports all our usage scenarios.

During our analysis, we notice that we have two interfaces that are empty – that is, they provide no additional functionality (other than typing). We may want to simplify this in the IT model. A refined Platform Independent Model (created based on the above discussion) is shown in Figure 5.

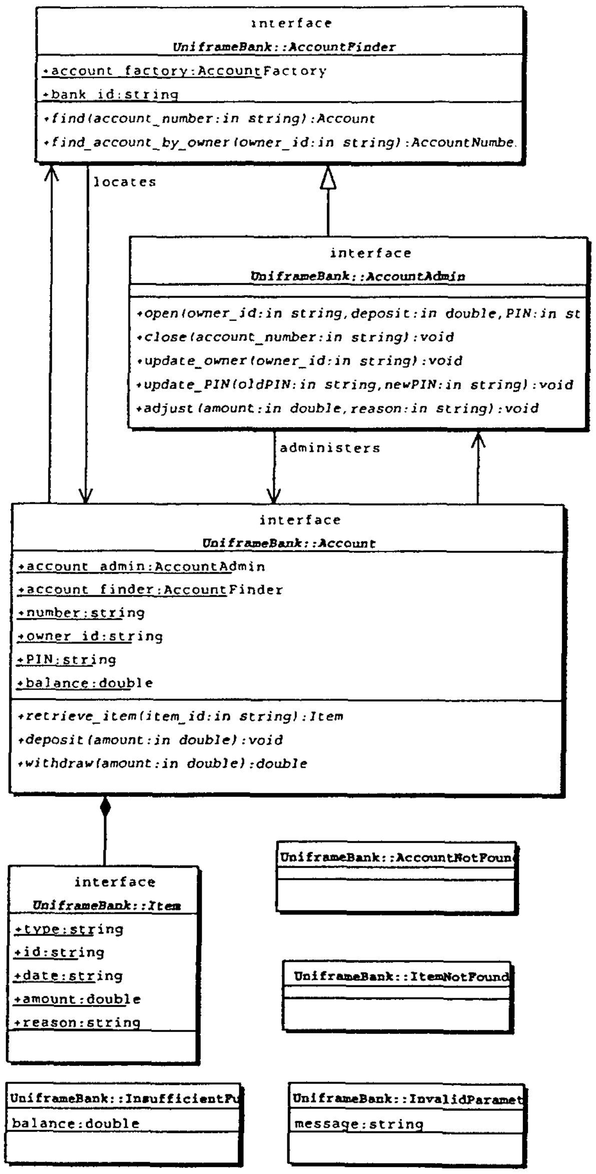

interface
UniframeBank::AccountFinder
+account_factory:AccountFactory
+bank_id:string
+find(account_number:in string):Account
+find_account_by_owner(owner_id:in string):AccountNumbe
locates
interface
UniframeBank::AccountAdmin
+open(owner_id:in string,deposit:in double,PIN:in st
+close(account_number:in string):void
+update_owner(owner_id:in string):void
+update_PIN(oldPIN:in string,newPIN:in string):void
+adjust(amount:in double,reason:in string):void
administers
interface
UniframeBank::Account
+account_admin:AccountAdmin
+account_finder:AccountFinder
+number:string
+owner_id:string
+PIN:string
+balance:double
+retrieve_item(item_id:in string):Item
+deposit(amount:in double):void
+withdraw(amount:in double):double
interface
UniframeBank::Item
+type:string
+id:string
+date:string
+amount:double
+reason:string
UniframeBank::AccountNotFoun
UniframeBank::ItemNotFound
UniframeBank::InsufficientFu
balance:double
UniframeBank::InvalidParamet
message:string

Figure 5: Simple Bank Platform Independent Model (PIM)

6.3 Simple Bank Paradigm Specific Model

The next step in the Model Driven Architecture is to find a way to use **all** the model information that has been captured in the use case analysis of the Platform Independent Model (PIM) and define the techniques that allow Platform Specific Models to be created that leverage all aspects of the PIM. We have reached the point where model optimizations must consider the characteristics of the target environment and/or platform.

As we make this transition, we see the value of progressing to a Platform Independent Model that is optimized for a particular computing paradigm; that is, the PIM may be used as a foundation for multiple platform specific implementations provided those platforms share some common characteristics. The characteristics or paradigms to consider include distributed solutions (distributed objects, synchronous messaging, asynchronous messaging, etc.), and local solutions (object-oriented programming, procedural programming, etc.). Other aspects such as embedded and/or real-time might also be considered at this time. A transition to a "paradigm specific model" is a useful intermediate step that captures the analysis necessary for a transition from a Platform Independent to Platform Specific Models. As such it may be useful in the development of algorithms that can be used with generative techniques for Platform Specific Models.

Are there usage scenarios that require remote access across wide area networks where network speed may be a factor?

Evaluate carefully each high usage remote access scenario for the following characteristics.

1. Does it require multiple network operations to accomplish what is logically a single request to the user?

Consider creating a service interface that offers services that wrap the existing service and gather all required information before responding.

2. Is it common to require and/or update multiple attributes simultaneously?

Consider passing structures or objects by value instead of using accessors and mutators on object attributes.

Figure 6: Experience-based remote access techniques

Continuing with our evaluation of quality of service issues, we focus on a distributed object paradigm as our technology choice. Figure 6 includes some of the techniques that we can use to refine our Platform Independent Model (these are illustrative and not exhaustive).

These design principles are key features of aspect oriented (or service oriented) architectures and are at the heart of what must be done for secure manageable web services. It is important to note that the business model is typically expressed as an object-oriented view of the business, not as a service oriented model. Therefore it is not possible to derive the service model directly from a business model with generative tools – that is, there is additional information (such as patterns of usage) that is not expressed in the business model that must be considered. The effect of this is that the resulting service model must be manually validated against the business model, as effects of changes on one model are not readily identifiable. This is a serious issue for business systems and one that will need to be addressed as MDA techniques and tools mature.

The analysis of our SimpleBank indicated that remote requests for account balance are very common and have a performance commitment associated with them. In addition, there were merchant services that have an impact on the revenue if performance commitments are not met. The PIM currently requires two independent requests across the network each time a balance is requested – first AccountFinder::find (account_number) to locate the account followed by Account::balance() to retrieve the balance. A more efficient remote operation (on some yet to be determined interface) might be get_balance (account_number) to allow this to be a single remote operation.

The key services of the bank are reflected in Figure 7 This is a paradigm specific model that is an addendum to the PIM. This reflects the requirement that distribution be considered and that key services be segmented for the purpose of performance enhancement, prioritization, and metrics.

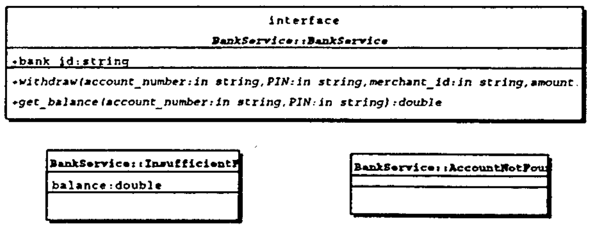

interface
BankService::BankService
+bank_id:string
+withdraw(account_number:in string,PIN:in string,merchant_id:in string,amount
+get_balance(account_number:in string,PIN:in string):double
BankService::InsufficientF
balance:double
BankService::AccountNotFou

Figure 7: Key Services Model for the SimpleBank

6.4 Simple Bank – Platform Specific Model

As we consider the technology platform that will be utilized, an evaluation of the quality of service requirements for the Simple Bank with regard to the platform features are part of the final transformation into a Platform Specific Model. These requirements (based on the analysis of the domain and the QoS parameters from our catalog) are:

Security: the service should be able to support dynamic decisions regarding exporting functionality to a user. The user should not be aware or have the ability to attempt to invoke any update operations unless they are authorized for update (that is, it should be possible at runtime to determine the interface offered to individual users).

Capacity: the system should be architected to scale to thousands of uses doing concurrent extensive work on hundreds of accounts. The usage is such that multiple operations will typically be done on accounts.

Maintainability: The ability to provide administrative services that extend the functionality must be available. The enhancement of these administrative interfaces should not impact the customers who are using the core features of the bank.

Performance: This is a distributed service. The service should be optimized for interactions across a wide area network at midrange speed.

The UniFrameBank designed to accommodate these requirements is defined below in ISO IDL. Note once again, that this interface model cannot be generated from the business model; a classic object-oriented design that does not take into consideration any QoS characteristics. The level of abstraction for the business model does not support the level of detail required to factor functionality in this way.

The UniFrameBank module defined in Figure 8 takes these QoS requirements and the usage scenarios into account. It introduces interfaces that respect the QoS requirements of the SimpleBanks' service offerings while maintaining the separation of concerns necessary to address security, ease of administration and maintainability.

```
module UniframeBank {

    typedef sequence<string> AccountNumbers;

    struct AccountInfo {
        string owner_id;
        string number;
        string PIN;
        double balance;
    };

    struct ItemInfo {
        string id;
        string type;
        string date;
        double amount;
        string reason;
    };

    exception AccountNotFound{};
    exception ItemNotFound{};
    exception InsufficientFunds{
       double balance;
    };
    exception InvalidParameter{
string message;
    };

  // Forward references
  interface AccountFactory;
  interface Account;
  interface AccountAdmin;

  // Key Services
  interface BankService
  {
    readonly attribute string bank_id;

    void withdraw(
            in string account_number,
            in string PIN,
            in string merchant_id,
            in double amount
    ) raises (
            AccountNotFound,
            InsufficientFunds
    );

    double get_balance(
            in string account_number,
            in string PIN
    )
            raises (AccountNotFound
    );
  };

  // AccountFinder
  interface AccountFinder {
    readonly attribute
             AccountFactory account_factory;
    readonly attribute string bank_id;

    Account find(
        in string account_number
    ) raises (
        AccountNotFound
    );

    AccountNumbers find_account_by_owner(
        in string owner_id
    ) raises (
        AccountNotFound
    );
  };
```

```
  // AccountFactory
  interface AccountFactory : AccountFinder {

    AccountInfo open(
        in string owner_id,
        in double deposit,
        in string PIN
    ) raises (
        InvalidParameter
    );

    void close(
        in string account_number
    ) raises (
AccountNotFound
    );
  };

  // Account
  interface Account {

    readonly attribute
           AccountAdmin  account_admin;
    readonly attribute
           AccountFinder account_finder;
    readonly attribute string number;
    readonly attribute string owner_id;
    readonly attribute string PIN;

    ItemInfo retrieve_item_info(
        in string item_id
    ) raises (
        ItemNotFound
    );

    void deposit (
          in double amount
    ) raises (
          InvalidParameter
    );

    double withdraw (
        in double amount
    ) raises (
        InsufficientFunds
       );
  };

  // AccountAdmin
  interface AccountAdmin :  Account   {
    void update_owner(
        in string owner_id
    ) raises (
        InvalidParameter
    );

    void update_PIN(
         in string oldPIN,
         in string newPIN
    ) raises (
         InvalidParameter
    );

    void adjust(
        in double amount,
        in string reason
    ) raises (
         InvalidParameter
    );
  };
};
```

Figure 8 – UniFrameBank – Platform Specific Model

An AccountFinder interface is responsible for locating accounts. This eases security because none of the operations on the Account are visible from this interface; hence if a client is not authorized to access an account they will be restricted from obtaining a reference to an Account object. In addition, the AccountFactory (which inherits from AccountFinder) is available only to clients who are authorized to open or close accounts. The AccountAdmin interface was introduced to allow evolution to more sophisticated services without affecting the interfaces of the coreaccount services (AccountFinder and AccountFactory). The client must be authorized to use an AccountAdmin object which had the ability to modify existing account attributes or items. The Account object offers only the core banking operations. The ability to request all Account or Item information in a single operation was added to the Account Interface to meet the performance requirements and limit the number of network interactions. The BankService interface that was introduced in the paradigm specific PIM is retained in the PSM.

7. Future Directions

The models and IDL presented in this paper will form the basis of the additional work to validate the experience-based design patterns presented in the paper and to progress techniques for the model parameterization with the goal of enabling generation of platform specific models such as those presented in this paper.

The next step in our research is to examine how these experienced-based patterns can be expressed as model parameters. We are hopeful that previous research (including our work on the QoS Catalog and TLG) [15] [16] and work in progress on standards for UML Profiles for QoS [14] can be leveraged. For this reason, we have not proposed a language for this purpose as yet. The QoS instrumentation is a complementary research activity. There is a need in component-based environments to progress instrumentation that can be utilized to determine whether a component can meet those QoS parameters when used within a composition. Of course, a part of the challenge is that the instrumentation introduces an additional overhead and in situations that are time sensitive or must be predictable, this overhead may disrupt the ability to measure the QoS parameter under observation. It is clear that a substantial amount of research needs to be done in this area and we plan to use an approach based on event grammars as indicated in [1] [17].

8. Conclusion

The ability to provide the QoS parameterization of models is recognized in the Object Management Group community and standards in this area will lead to the ability to generate Platform Specific Models that take quality of service characteristics into account. However, since there has been a very little work on progressing Quality of Service specifications for component-based architectures, UniFrame research has a potential to impact how the Object Management Group (OMG) defines QoS parameterization for Model Driven Architecture and the ability to more clearly specify and measure component feasibility for a particular task. The standardization of QoS catalogs and parameters is a pre-requisite to defining algorithms for the transformation of Platform Independent Models into Platform Specific Models. In addition, benchmarking and service validation via instrumentation require that such standards exist. Our expectation is that any Quality of Service parameters defined by OMG will be applicable for CORBA®, J2EE™, and Web Services component architectures.

Quality of Service characteristics must have syntax for expression in every artifact of the analysis, design and development process. Design patterns must be documented and exploited in such a way that generative techniques can be applied. In addition, formal specifications will allow the instrumentation necessary for measuring quality of service to be come an integral part of middleware and component implementation frameworks.

QoS-oriented software development is of paramount importance to delivering robust, scalable and secure distributed component solutions.

9. References

[1] Rajeev R. Raje, Barrett Bryant, Mikhail Auguston, Andrew Olson, Carol Burt. "*A Unified Approach for the Integration of Distributed Heterogeneous Software Components*", Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration, pp: 109-119, Monterey, California, 2001.

[2] Object Management Group. 2001. *Model Driven Architecture: A Technical Perspective*. Technical Report. Document # ormsc/2001-07-01. Framingham, MA: Object Management Group. July 2001.

[3] Carol C. Burt, Barrett R. Bryant, Rajeev R. Raje, Andrew Olson. Mikhail Auguston. 2002. *Quality of Service (QoS) Standards for Model Driven Architecture.* Proceedings of the 2002 Southeastern Software Engineering Conference (to appear).

[4] K. Czarnecki, U. W. Eisenecker, 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley.

[5] Object Management Group. 2001. *CORBA 3.0 CORBA Component Model Chapters.* Document # ptc/2001-11-03. Framingham, MA: Object Management Group.

[6] Sun Microsystems. 2001. *Java 2 Platform Enterprise Edition Specification v1.3*, Available via ftp from www.java.sun.com. Sun Microsystems.

[7] Rajeev R. Raje, Mikhail Auguston, Barrett Bryant, Andrew Olson, Carol Burt. 2001. *A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components*. Technical Report. Indiana University Purdue University Indianapolis.

[8] J. A. Zinky, D. E. Bakken, R. Schantz,, 1995. *Overview of Quality of Service for Distributed Objects*, Proceedings of the Fifth IEEE Dual Use Conference.

[9] S. Frolund, J. Koistinen. 1998. *Quality of Service specification in Distributed Object Systems*, Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98).

[10] S. Frolund, J. Koistinen. 1999. *Quality of Service Aware Distributed Object Systems.* 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99).

[11] Ringo Ling, Hugo Latapie, Vu Tran, 2002. *Expressing Common Criteria Security Requirements in Domain Models in Model-base Architecture.* Technical Presentation. Distributed Object Security Conference (DocSec 2002). Baltimore, MD. March 2002.

[12] BBN Corporation, 2001. *Quality Objects (QuO) Project*, URL: http://www.dist-systems.bbn.com/tech/QuO.

[13] Chris Jones, Partha Pal, Franklin Webber, 2002. *Defense Enabling Using QuO: Experience in Building Survivable CORBA Applications.* Technical Presentation. Distributed Object Security Conference (DocSec 2002). Baltimore, MD. March 2002.

[14] Object Management Group. 2002. *UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms.* Request for Proposal. OMG document ad/02-01-07. Framington, MA. Note: This RFP issued January 2002 with submissions due June 24, 2002.

[15] Girish J. Brahnmath, Rajeev R. Raje, Andrew M. Olson, Mikhail Auguston, Barrett R. Bryant, Carol C. Burt. 2002. *A Quality of Service Catalog for Software Components.* Proceedings of the 2002 Southeastern Software Engineering Conference (to appear).

[16] Barrett Bryant, Mikhail Auguston, Rajeev R. Raje, Andrew M. Olson, Carol C. Burt. 2002. *Formal Specification of Generative Component Assembly using Two-Level Grammar.* Technical Report. University of Alabama Birmingham.

[17] Mikhail Auguston. 2000. *Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars.* Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000), pp.159-166.

[18] Object Management Group. 2000-2002.*OMG Adopted Technology for UML, UML Profiles, Meta Object Facility and Common Meta-Data Warehouse.* These OMG documents are available from OMG via http://www.omg.org/technology/documents/modeling_spec_catalog.htm. Framingham, MA: Object Management Group.

A Framework for Automatic Debugging

Mikhail Auguston, Clinton Jeffery, Scott Underwood
Department of Computer Science, New Mexico State University
{mikau, jeffery, sunderwo}@cs.nmsu.edu

Abstract

This paper presents an application framework in which declarative specifications of debugging actions are translated into execution monitors that can automatically detect bugs. The approach is non-intrusive with respect to program source code and provides a high level of abstraction for debugging activities.

1. Motivation

Debugging is one of the most challenging, and least developed areas of software engineering. Debugging activities include queries regarding many aspects of target program behavior: sequences of steps performed, histories of variable values, function call hierarchies, checking of pre- and post-conditions at specific points, and validating other assertions about program execution. Performance testing and debugging involves a variety of profiles and time measurements.

We are building automatic debugging tools based on precise program execution behavior models that enable us to employ a systematic approach. Our program behavior models are based on events and event traces [1][2][3].

Debugging automation refers to a computation over an event trace. *Program execution monitors* are programs that load and execute a target program, obtain events at run-time, and perform computations over the event trace. Computations are performed during execution, post-mortem, or in any mixture of both times.

Any detectable action performed during a target program's run time is an *event*. For instance, expression evaluations, statement executions, and procedure calls are all examples of events. An event has a beginning, an end, and some duration; it occupies a time interval during program execution. This leads to the introduction of two basic binary relations on events: partial ordering and inclusion. Those relations are determined by target language syntax and semantics, e.g. two statement execution events may be ordered, or an expression evaluation event may occur inside a statement execution event. The set of events produced at the program run time, together with ordering and inclusion relations, is called an *event trace* and represents a model of program behavior. An event trace forms an acyclic directed graph (DAG) with two types of edges corresponding to the basic relations.

The language UFO (from Unicon-FORMAN) integrates the experience accumulated in the FORMAN [1] language and the Alamo monitoring architecture [4] to provide a complete solution for development of an extensive suite of automatic debugging tools. UFO is an implementation of FORMAN for debugging programs written in the Unicon and Icon programming languages [5][6].

2. Unicon and Alamo

Unicon is an imperative, goal-directed, object-oriented superset of Icon. Unicon's syntax is similar to Pascal or Java; its semantics features built-in backtracking, heterogeneous data structures and string scanning facilities. Unicon extends Icon's reach with elegant object-orientation, high level networking, messaging, and database facilities.

The reference implementation of Unicon is a virtual machine. Virtual machines (VMs) are attractive to language implementers because they provide portability and a vastly simpler implementation of very high level language features such as backtracking. As a result, event detection is an integral part of the VM.

VMs are ideal for developing debugging tools; they provide an appropriate level of abstraction for behavior models that describe program executions in a processor independent manner, as illustrated by the JPAX tool [7].

In Alamo, monitors and the target program execute as (sets of) coroutines with separate stacks and heaps inside a common VM. The Unicon VM is instrumented with over 100 kinds of atomic events, each one capable of reporting a <code,value> pair to monitors with interest in that event. Event reports are coroutine context switches.

Monitors are written independently from the target program, and can be applied to any target program without recompiling the monitor or target program. Monitors dynamically load target programs, and can easily query the state of arbitrary variables at each event report. Multiple monitors can monitor a program execution, under the direction of a monitor coordinator.

Alamo's goal was to reduce the difficulty of writing execution monitors to be just as easy as writing other types of application programs. UFO supports FORMAN's more ambitious goal of reducing the task of writing automatic debuggers to the task of specifying generic assertions about program behavior.

3. An Event Grammar for Unicon

Event grammars provide a model of program run time behavior. Monitors do not have to parse events using this grammar, since event detection is part of VM and UFO runtime system functionality. The following description provides a "lightweight" semantics of the Unicon programming language tailored for specification of debugging activities.

An event corresponds to a specific action of interest performed during program execution. Each event has one or more types and related attributes associated with it.

Universal attributes are found in every event. They are frequently used to narrow assertions down to a particular domain (function, variable, value) of interest. Some of the universal attributes are:

source_text: in canonical form (i.e. with redundant spaces eliminated, etc.)
line_num, col_num: source text locations
time_at_end, time_at_begin, duration: timing attributes
value_at_begin (Unicon-expression),
value_at_end (Unicon-expression): these attributes provide access to the program states

The event types, and type-specific attributes they provide, are summarized in the table below.

Event Type	Description	Attributes
prog_ex	whole program execution	
expr_eval	expression evaluation	value, operator, type, failure_p
func_call	function call	name, paramlist
param	actual parameter evaluation	name
func_body	function body execution	
input, output	I/O	file
variable	variable reference	
literal	reference to a constant value	
lhp	lefthand part, assignment	address
rhp	righthand part, assignment	
clause	then-, else-, or case branch execution	
test	test evaluation	
iteration	loop iteration	
return	return from procedure call	

Event types form a class hierarchy, shown in Figure 1. Subtypes inherit attributes from the parent type.

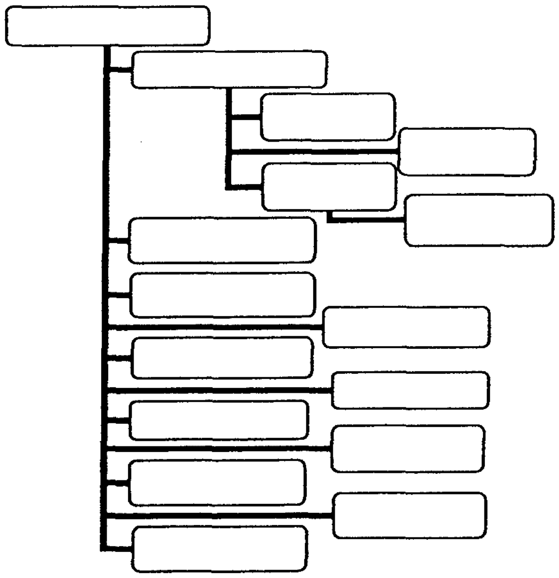


Figure 1. Event Type Inheritance Hierarchy

The UFO *event grammar* for Unicon is a set of axioms describing the structure of event traces with respect to two basic relations: inclusion and precedence. The grammar shown below is one possible abstraction of Unicon semantics; other event grammars might be used. The event grammar limits what kinds of bugs can be detected, so detail is useful. The grammar uses the notation:

Notation	Meaning
A :: (B C)	B precedes A, A includes B and C
A*	Zero or more A's under precedence
A+	One or more A's under precedence
A | B	Either A or B; alternative
A?	A is optional
{ A , B }	Set; A and B have no precedence

```
prog_ex::   ( expr_eval *)
expr_eval::( ( expr_eval ) |                 unary op
             ( expr_eval  expr_eval ) |      binary op
             ( expr_eval+ ) |
             ( test  clause ) |              conditional/
                                          case expressions
             ( iteration * ) |               loops
             ( { lhp,  rhp} ) )              assignment
```

```
                   lhp and rhp are not
                   ordered, beginning of
                   lhp precedes rhp, and
                   end of lhp follows rhp
iteration::        ( test expr_eval*) | ( expr_eval* test ) |
                   ( expr_eval * )
func_call::        ( param* func_body )
func_body::        ( expr_eval* return? )
```

Execution of a Unicon program produces an *event trace* organized by precedence and inclusion into a DAG. The structure of the event trace (event types, precedence and inclusion of events) is constrained by the event grammar axioms above. The event trace models Unicon program behavior and provides a basis to define debugging activities (assertion checking, debugging queries, profiles, debugging rules, behavior visualization) as appropriate computations over the event traces.

4. FORMAN

Alamo allows efficient monitors to be constructed in Unicon, but using a special-purpose language such as FORMAN, with the rich behavior model described in the preceding section, has compelling advantages. For example, in FORMAN we may refer to target program variable x, while in the Unicon monitor it is referenced as variable("x", &eventsource).

More important than such notational conveniences are FORMAN's control structures that support computations over event traces, centered around the notions of event pattern and aggregate operations over events.

The simplest event pattern comprises just an event type and matches successfully an event of this type or an event of a subtype of this type. Event patterns may include event attributes and other event patterns to specify the context of an event under consideration. For example, the event pattern

```
E: expr_eval:: ( R: rhp & is_an_object(R.value))
                  & E.operator == ":="
```

matches an event of assignment type where the right hand part evaluates to an object. Temporary variables E and R provide an access to the events under consideration within the pattern.

The following example demonstrates the use of an aggregate operation.

```
CARD[ A: func_call &
            A.func_name == "read" FROM prog_ex ]
```

yields a number of events satisfying given event pattern, collected from the whole execution history. Expression [...] is a list constructor and CARD is an abbreviation for a reduction of '+' operation over the more general list constructor:

```
+/[A: func_call & A.func_name == "read"
  FROM prog_ex APPLY 1 ]
```

Quantifiers are introduced as abbreviations for reductions of Boolean operations OR and AND. For instance,

```
FOREACH Pattern FROM event_set Boolean_expr
```

is an abbreviation for

```
AND/[Pattern FROM event_set APPLY Boolean_expr ]
```

Debugging rules in FORMAN usually have the form:
Quantified_expr SAY-clauses ONFAIL SAY-clauses
The Quantified-expr is optional and defaults to TRUE. The execution of FORMAN programs relies on the Unicon monitors embedded in a virtual machine environment.

5. Examples of Debugging Rules

UFO supports and improves upon the most common application-specific debugging techniques. For example, UFO supports traditional precondition checking, or print statement insertion, without any modification of the target program source code. This is useful when the precondition check or print statement is needed in many locations scattered throughout the code.

Example #1: Tracing. Probably the most common debugging method is to insert output statements to generate trace files. It is possible to request evaluation of arbitrary Unicon expressions at the beginning or at the end of events.

```
DO AT EVERY  A: func_call &
        A.func_name == "my_func"
FROM prog_ex {
  BEFORE A
     { write("entering my_func, value of X is:", X) }
  AFTER A
     { write("leaving my_func, value of X is:", X) }
}
```

This debugging rule causes run time instrumentation with calls to write() at selected points, before and after each occurrence of event A.

Example #2: Profiling. A myriad of tools are based on a premise of accumulating the number of times a behavior occurs, or the amount of time spent in a particular activity or section of code. The following debugging rule comprises several computations over the event trace.

```
SAY("Total number of read() statements: "
     CARD[ r:input & r.filename == "xx.in"
              FROM prog_ex ]
       "Elapsed time for read operations is: "
       +/ [ r:input & r.filename=="xx.in"
              FROM prog_ex APPLY r.duration]
```

Another interesting prospect is the development of a suite of generic automated debugging tools that can be used on any Unicon program. UFO provides a level of abstraction sufficient for specifying typical bugs and debugging rules. So far, the automatic debugging encyclopedia at http://www.cs.nmsu.edu/please/bugs.html has entries for 53 common bugs.

Example #3: Detecting Use of Un-initialized Variables. Reading an un-initialized variable is allowed in Unicon, but often leads to errors. Therefore, in this debugging rule all variables within the target program are checked to ensure that they are initialized before they are used.

```
FOREACH E: expr_eval CONTAINS (V: variable)
FROM prog_ex
  EXISTS D: lhp FROM E.prev_path
    D.source_text == V.source_text AND
      V.source_text BELONGS_TO
        ( E.scope SCOPE_INTERSECTION D.scope )
  ONFAIL SAY("Expression" E " contains the "
                " uninitialized variable " V.source_text)
```

SCOPE_INTERSECTION is similar to a set intersection, except that it takes into account scoping and visibility rules of the source language.

Example #4: Closed Files. Failure to close files that have been opened is an easily overlooked error. This assertion detects this event and warns the user. The temporary variable NumberOfClose holds the cardinality of the close() event set.

```
FOREACH a: func_call::(b:param) &
                a.func_name == "open"
  LET NumberOfClose =
    CARD[c:func_call::(d:param) &
          c.func_name == "close" &
          b.source_text == d.source_text]
  IN IF NumberOfClose == 0 THEN
      SAY("Failed to close file" b.source_text
                "after opening at event " ,a)
      ELSEIF NumberOfClose > 1 THEN
        SAY("Attempt to close file " b.source_text
              "more than once") ENDIF
```

6. Implementation Issues

This section describes issues that have arisen during the implementation of UFO. The most important of these issues is the translation model by which FORMAN assertions are compiled down to Unicon Alamo monitors. Debugging activities are written as if they have the complete post-mortem event trace, the DAG with events, precedence and containment relations, available for processing. This generality is extremely powerful; however the vast majority of assertions can be compiled down into monitors that execute entirely at runtime. Runtime monitoring saves enormously on memory and I/O requirements and is the key to practical implementation. For those assertions that require post-mortem analysis, the UFO runtime system will compute a projection of the execution DAG necessary to perform the analysis.

The first step in generating code under the UFO translation model is to categorize each assertion as either "runtime", "post-mortem", or "hybrid", denoting the extent to which that assertion can be performed at runtime. Runtime and hybrid categorization is determined by constraints on FORMAN quantifier prefixes and results in more efficient monitor code. Nested quantifiers generally require post-mortem operation.

The UFO compiler generates Alamo Unicon monitors from FORMAN rules. Each FORMAN statement is translated into a combination of initialization, run-time, and post-mortem code. Monitors are executed as coroutines with the Unicon target program.

Implementation of Example #1: Tracing. A single DO AT EVERY quantifier is quite typical of many UFO debugging actions and allows computation to be performed entirely at runtime. The events being counted and values being accumulated are used to construct an *event mask* in the initialization code that defines the Alamo events that will be monitored.

The monitor's event processing loop implements the filter based on procedure name within an if-expression. The Unicon code blocks containing write() expressions are inserted directly into the event loop for the relevant events. The complete monitor is:

```
$include "evdefs.icn"
link evinit
procedure main(av)
  EvInit(av) | stop("can't monitor ", av[1])

  ### initialization for BEFORE and AFTER func_call
  mask:= E_Pcall ++ E_Pret ++ E_Pfail

  while EvGet(mask) do {
    if &eventcode == E_Pcall &
      image(&eventvalue)=="procedure my_func" then
        ### inserted BEFORE clause
        write("entering my_func, value of X is:",
               variable("X", Monitored))
    if &eventcode == (E_Pret | E_Pfail) &
      image(&eventvalue)=="procedure my_func" then
       ### inserted AFTER clause
       write("leaving my_func, value of X is:",
            variable("X", Monitored))
      }
   end
```

Implementation of Example #2: Profiler. This is another typical situation, which involves an aggregate operation and selection of events according to a given pattern. The SAY expression is implemented by a call to write(); it must be performed post-mortem since it uses parameters whose values are constructed during the entire program execution. CARD denotes a counter, while SUM denotes an accumulator +/; both require a variable that is initialized to zero. The event subtypes and constraints are used to generate additional conditional code in the body of the event processing loop. Lastly, some attributes such as r.duration require additional events and measurements besides the initial triggering event. In the case of r.duration, a time measurement between the function call and its return is needed.

```
$include "evdefs.icn"
link evinit
procedure main(av)
   EvInit(av) | stop("can't monitor ", av[1])
  ### initialization for CARD and SUM
  cardreads := 0
  sumreadtime := 0
  mask := E_Fcall
  while EvGet(mask) do {
    ### count CARD of r:input...
    if &eventcode == E_Fcall &
      &eventvalue === (read|reads) then
       cardreads +:= 1
    ### add SUM of r.duration for r:input
    if &eventcode == E_Fcall &
      &eventvalue === (read|reads) then {
       thiscall := &time
       EvGet(E_Ffail++E_Fret)
       sumreadtime +:= &time - thiscall
      }
    }
  ### Translation of SAY
  write("Total number of read() statements: ",
        cardreads, "\n",
       "Elapsed time for read operations is: ",
        sumreadtime)
end
```

The advantage of the UFO approach is the combination of an optimizing compiler for monitoring code with efficient run-time event detection and reporting. Since we know at compile time all necessary event types and attributes required for a given FORMAN program, the generated Unicon monitor can be very selective about the behavior that it observes. The compiler merges several computations such as operation reduction or quantifiers present in the FORMAN assertions into a single Unicon event loop. Since the compiler processes several assertions together, it can merge overlapping constructs (for example, those referring to the same events).

For certain kinds of FORMAN constructs, such as nested quantifiers, the monitor must accumulate a sizable projection of the complete event trace and postpone corresponding computations until all required information is available, and schedule corresponding computations. The most challenging and interesting remaining part of this compilation effort is to further optimize this analysis.

UFO's goal of practical application to real-sized programs has motivated improvements to the Alamo instrumentation of the Unicon VM. Although UFO is not complete enough to report conclusive results, the following table illustrates the effects of certain optimizations. The program in question is a mail message indexing tool, which processes mail headers and builds indices. For test purposes it is executed on a sample input of 3MB. All results are in seconds. The leftmost column shows the application's normal runtime. Columns 2-5 show runtimes for Implementation Example #2 above (the I/O function profiler) under Alamo, and three levels of optimization under UFO. Alamo imposed a 200% slowdown for comprehensive VM instrumentation, plus less than 100% slowdown for monitor code. Very little of the VM instrumentation is actually needed for this example. UFO-IO shows the effect of instrumentation optimization which UFO does at compile-time, optionally generating a custom VM for a given suite of FORMAN assertions. UFO-CO shows additional compiler optimizations on the monitor code. UFO-VM shows the effect of a runtime optimization called *value masking* on the virtual machine instrumentation. We are working on additional optimizations, and believe the end result will be highly practical execution from our high-level framework.

No monitor	Alamo	UFO-IO	UFO-CO	UFO-VM
1.35	3.64	2.82	2.30	1.87

7. Related Work

See www.cs.nmsu.edu/TechReports/2002/004.pdf for an expansion of this survey of related work.

The Event Based Behavioral Abstraction (EBBA) [8] characterizes program behavior in terms of primitive and composite events. Dalek is an event-based debugger for C built on top of GDB [9].

FORMAN takes a more comprehensive modeling approach than EBBA or Dalek, based on an event grammar and a language for expressing computations over execution histories. Event grammars make FORMAN suitable for automatic source code instrumentation. FORMAN's abstraction of event as a time interval provides an appropriate level of granularity for reasoning about behavior, in contrast with the event notion in previous approaches where events are considered point-wise time moments.

Monitoring frameworks such as Dalek and COCA [10] use GDB to attain a necessary level of abstraction, which UFO finds in the Unicon virtual machine. While both approaches yield adequate source-level access and control over the monitored program, the virtual machine approach avoids substantial operating system overhead and offers better performance and scalability to larger programs.

Assertion languages provide yet another approach to debugging automation. Most approaches are based on Boolean expressions attached to points in the target program, like the assert() macro in C. [13,14,15] give approaches to programming with assertions for C and Ada. Even local assertions associated with particular points within the program may be extremely useful for program debugging. The DUEL [11] debugging language introduces expressions for C aggregate data exploration, for both assertions and queries.

The notion of computation over execution trace introduced in FORMAN is a generalization of Algorithmic Debugging [21, 22] and may be a convenient basis for describing generic debugging strategies.

PMMS [12] receives queries about target programs written in AP5, instruments source code, and stores data in a database to answer the posed questions. PMMS's domain specific query language is similar to FORMAN but tailored for database-style query processing.

8. Conclusions

The popularity of virtual machines promises to enable dramatic improvements in automatic debugging. These improvements will only occur if debugging is a specific goal of the virtual machine, e.g. as in the case of .net [13].

UFO illustrates what is possible for a broad class of languages such as those supported by the Java VM or the .net VM. Our approach uniformly represents many types of debugging-related activities as computations over traces. We have shown an approach to integrating event trace computations into a monitoring architecture based on a virtual machine. The end result provides a suitable environment for the implementation of automated debugging tools.

Acknowledgements

This work has been supported in part by U.S. Office of Naval Research Grant # N00014-01-1-0746, by U.S. Army Research Office Grant # 40473--MA-SP, and by the National Library of Medicine.

References

[1] Mikhail Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, in Proceedings of AADEBUG'95, Saint-Malo, France, May 22-24, 1995, pp. 277-291.

[2] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", in Proceedings of SEKE'97, Madrid, Spain, June 1997, pp. 257-262.

[3] M. Auguston, "Lightweight semantics models for program testing and debugging automation", in Proceedings of the 7th Monterey Workshop on "Modeling Software System Structures in a Fast Moving Scenario", Santa Margherita Ligure, Italy, June 13-16, 2000, pp.23-31.

[4] Clinton L. Jeffery, Program Monitoring and Visualization: an Exploratory Approach. Springer, New York, 1999.

[5] Clinton Jeffery, Shamim Mohamed, Ray Pereda, and Robert Parlett, "Programming with Unicon", http://unicon.sourceforge.net.

[6] Ralph E. Griswold and Madge T. Griswold, The Icon Programming Language, 3rd edition. Peer to Peer Communications, San Jose, 1997.

[7] K. Havelund, S. Johnson, and G. Rosu. "Specification and Error Pattern Based Program Monitoring", European Space Agency Workshop on On-Board Autonomy, Noordwijk, Holland, October 2001.

[8] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", The Journal of Systems and Software 3, 1983, pp. 255-264.

[9] R. Olsson, R. Crawford, W. Wilson, "A Dataflow Approach to Event-based Debugging", Software -- Practice and Experience, Vol.21(2), February 1991, pp. 19-31.

[10] M. Ducasse, "COCA: An automated debugger for C", in Proceedings of ICSE 99, Los Angeles, 1999, pp.504-513.

[11] M. Golan, D. Hanson, "DUEL - A Very High-Level Debugging Language", in Proceedings of the Winter USENIX Technical Conference, San Diego, Jan. 1993.

[12] Y. Liao, D. Cohen, "A Specificational Approach to High Level Program Monitoring and Measuring", IEEE Transactions On Software Engineering, Vol. 18, No. 11, November 1992, 969 – 978.

[13] http://www.microsoft.com/net/.

Unified Approach for System-Level Generative Programming

Zhisheng Huang, Rajeev R. Raje,
Andrew M. Olson
Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202, USA
{zhuang, rraje, aolson, csun}@cs.iupui.edu,
+1 317 274 5246/5174/9733

Mikhail Auguston
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
mikau@cs.nmsu.edu, +1 505 646 5286

Barrett R. Bryant, Carol Burt
Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, Alabama 35294-1170, USA
{bryant, cburt}@cis.uab.edu, +1 205 934 2213

Changlin Sun
Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202, USA
csun@cs.iupui.edu, +1 317 274 5246

Abstract

Today's and future distributed software systems will certainly require combining heterogeneous software components that are geographically dispersed so that its realization not only meets the functional requirements, but also satisfies the non-functional criteria such as the desired quality of services (QoS). The Unified Approach (UA) incorporates the concepts of product line practice (PLP) and generative programming with the Unified Meta-component Model (UMM) to achieve automatic development, maximal reuse and seamless interoperation. The creation of a software solution for a distributed computing system (DCS), using the UA has two levels, the component level and the system level. In this paper, the system-level generative programming of the UA is described.

Keywords: Distributed Computing Systems, Heterogeneous Components, Quality of Services, Generative Programming, Generative Domain Model, Two-Level Grammar.

1. Introduction

As distributed computing becomes more and more crucial for the success of today's enterprises, there is an increasing need to develop software for a distributed computing system (DCS) in an effective and efficient way. A lot of distributed computing systems are still designed and built as single systems. This approach has the problems of large investment, long development cycles, difficulties in the system integration, and a lack of predictable quality. Generative programming [7] and product line practice (PLP) [19] help us to move the focus from the development of single systems to system families. The use of components to develop software for a DCS is consistent with the notions of generative programming and PLP. However, another challenge arises as component-based software development is applied to distributed computing. This challenge is an effect of the presence of multiple component models. Currently, different component models have been proposed, such as Java™ Remote Method Invocation (RMI) [13], Common Object Request Broker Architecture (CORBA™) [11, 13, 17], and the Distributed Component Object Model (DCOM™) [10]. There are difficulties in bridging the components of different models, thus reducing the component reuse. The Unified Meta-Component Model Framework (UniFrame) research [14, 15, 16] is an attempt to unify the existing and emerging distributed component models under a common meta-model, the Unified Meta-component Model (UMM), for the purpose of enabling the discovery, interoperability and collaboration of components via a Unified Approach (UA). The UA is a UMM-based technique, which incorporates some ideas from generative programming and PLP. It replaces the manual search for, and adaptation and assembly of, heterogeneous and distributed components with automation. The aim is to develop a quality-oriented and time-to-market DCS with lower development and maintenance costs. The creation of a software realization of a DCS using the UA has two levels: a) the component level – component development and deployment, and b) the system level – automatic or semiautomatic system generation.

This paper describes the UA at the system level. The principles of generative programming, PLP and the UniFame are briefly described in the next section. Section 3 discusses, in detail, the system-level generative programming of the UA, which is illustrated by an example in section 4. The paper concludes in section 5.

2. Related work

2.1 Generative programming

The generative programming is concerned with bringing automation to the software development. In [7] the generative programming paradigm is defined as: "Generative Programming is about manufacturing software products out of components in an automated way. It requires two steps: a) a design and implementation of a generative domain model, representing a family of software systems (development for reuse). This model includes also a domain-specific software generator; b) given a particular requirements specification, a highly customized and optimized end-product can be automatically manufactured from implementation components by means of generation rules (development with reuse)". The methods presented in [7] can be applied both "in the small", i.e., at the level of classes and procedures and "in the large", to develop families of large systems.

2.2 PLP

In 1997, the PLP initiative [19] was launched by the Software Engineering Institute (SEI) of Carnegie Mellon University. The intention was to help facilitate and accelerate the transition from the traditional single system development to sound software engineering practices using a product line approach. A software product line is defined to be a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a selected market or mission, and that are developed from a common set of core assets in a prescribed way [5, 6]. The SEI's PLP Framework is the first formal attempt to codify the comprehensive information about successful product lines. The idea behind this framework is to identify the different issues and practices relevant to establishing and running successful product lines in an organization. More information can be found on the PLP Framework website [20].

2.3 UniFrame

The UniFrame provides a framework for constructing a DCS by integrating the heterogeneous and distributed software components. It consists of the Unified Meta-component Model (UMM) and the Unified Approach (UA).

2.3.1 UMM. The recent shift in the focus of Object Management Group (OMG) to Model Driven Architecture (MDA) [12] is a recognition that bridging components to create DCS requires standardization of not only the infrastructure but also Business and Component Models. The UMM provides an opportunity to bridge gaps that currently exist in the standards arena. The core parts of the UMM are: components, service and service guarantees, and infrastructure. In UMM, components are autonomous entities. All components have well-defined interfaces and private implementation. In addition, each component in UMM has three aspects: a) computational aspect, b) cooperative aspect, and c) auxiliary aspect. Each component must be able to specify and guarantee the quality of service (QoS) offered. The headhunter [18] and Internet Component Broker (ICB) [15, 18] of the infrastructure are responsible for allowing a seamless integration of different component models and sustaining cooperation among heterogeneous components (adhering to different models). The headhunter is responsible for searching and managing heterogeneous and geographically distributed components. The ICB acts as a translator between two heterogeneous components. An ICB itself is a component defined under the UMM. It achieves interoperability using the principles of wrap and glue technology [9]. An example of ICB is a Java – CORBA bridge, which bridges a component of Java RMI technology and a component of CORBA technology. For a detailed description of UMM, see [14, 15, 16].

2.3.2 UA. The UA is the UMM-based technique for the automatic production of a DCS. The creation of a software realization of a DCS using UA has two levels: a) the component level - components are designed and developed with UMM specifications (which are informal in nature [14]), tested and validated against the appropriate QoS, then deployed on the network, and b) the system level – a semi-automatic or automatic generation of a specific DCS product from a DCS family. The concepts of generative programming are applied at both levels in the UA. This paper describes the application of generative programming at the system level.

3. System-level generative programming of the Unified Approach (UA)

3.1 Core activities in the UA

The UA has four core activities for building distributed systems. These are: *generative domain engineering*, *component engineering*, *generative application engineering*, and *active distributed component management*. Their relationships are depicted in Figure 1. The development process is iterative and there are feedbacks during the first three activities. These four core activities span both the levels of UA: the component level and the system level. The first two activities, *generative domain engineering* and *component engineering*, corresponding to the domain engineering in [7], aim at maximizing the reuse of both the components and the software architecture. We distinguish between these two activities because they reflect the different levels in the UA. *Generative domain engineering* is a system-level activity and the *component engineering* is at the component level. *Generative application engineering* is another system-level activity. *Active distributed component engineering* is involved at both levels.

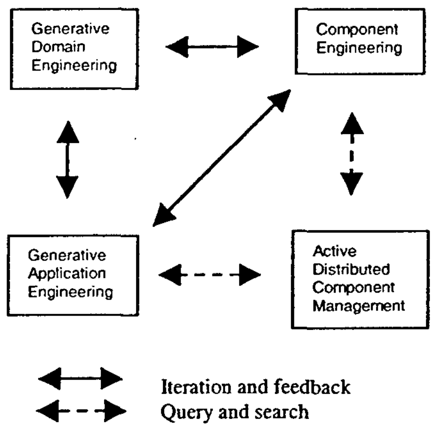

Generative Domain Engineering
Component Engineering
Generative Application Engineering
Active Distributed Component Management
Iteration and feedback
Query and search

Figure 1. UA core activities

Generative domain engineering consists of activities for identifying commonalities and variations of the system architecture of a DCS family. It is responsible for creating the generative domain model (GDM), which is discussed in 3.2, to represent a configurable system architecture. This architecture includes a set of abstract components as the guidelines for developing reusable concrete components during *component engineering* phase. Each abstract component represents one component type and is defined with its UMM specification. This specification is natural language-like and includes both the functional and nonfunctional (such as expected QoS properties) aspects of a component [14]. This specification is then refined into a formal specification, based upon the theory of Two-Level Grammar (TLG) [4] and natural language specifications [3]. The GDM is the core software asset that results from *generative domain engineering*.

During *component engineering* phase, the abstract components are mapped to different component models to create concrete components. The concrete components are tested and validated against the appropriate QoS, deployed over the network, and then are discovered by the headhunters. It is worthwhile to note that the generative programming is also carried out in the *component engineering* phase of the UA.

Generative application engineering is the process of building a DCS based on a GDM. It is supported by the query processor (see explanation in section 3.4) and *active distributed component management*. During *generative application engineering*, a DCS is produced out of a DCS family in three steps: a) determining the target system and its architecture instance according to the system specification produced by the query processor; b) searching for concrete components for the target system via the headhunter; and c) assembling and testing the DCS according to the architecture instance to produce a workable distributed system that meets both the functional and non-functional requirements. The GDM is used to guide the system assembly and validation. The validation of the QoS requirements is carried out both by QoS composition rules [21], which specify how the system QoS or subsystem QoS can be composed from the QoS of its parts, and by the event grammars [1, 2], which are used as the basis for the system behavior models to trace events like executing a statement or calling a procedure. The example in the next section illustrates these steps.

Active distributed component management is the UniFrame resource discovery service (URDS) [18]. It offers the dynamic discovery and management of the heterogeneous software components and assists in the finding of the required components during the phase of the *generative application engineering*. These are achieved by headhunters, which are analogous to binders or traders in other models, with one difference - a trader is passive, while a headhunter is active. For details, see [18].

3.2 UA GDM

The key to automating the manufacturing of systems is a GDM, which consists of a problem space, a solution space, and the configuration knowledge mapping between them [7]. The problem space consists of the application-oriented concepts and features that application developers can use to express their needs. UA GDM contains a Design Space Model (DSM) to represent the common and variable properties of a software architecture and a set of abstract components as guidelines for creating reusable distributed components. The DSM is an important part of the problem space. DSM describes the configurable software architecture with feature notations as described in [7], but, additionally, classifies the architectural nodes that are divided into five types: *domain*, *system*, *subsystem*, *design* and *abstract component*. In the graphical representation of the GDM, these node types are represented by surrounding the name of each node with << >>, < >, ( ), { }, and [ ] respectively. Associated with each node type is a standardized description, such as the UMM description for an abstract component. With the introduction of node types, a configurable system architecture can be easily represented. The description associated with a node shows information such as the relationship between its constituents (its children in the DSM). A simple example of a DSM is described in the next paragraph. The solution space consists of concrete components developed during component engineering when abstract components are mapped to specific component models and implemented. The configuration knowledge includes, as stated in [7], illegal feature combinations, default settings, default dependencies, construction rules and optimization rules, etc. In UA, it also includes additional important knowledge, such as, QoS composition and decomposition rules [21], which help ensure the assembled distributed system meets not only the functional requirements but also the non-functional requirements.

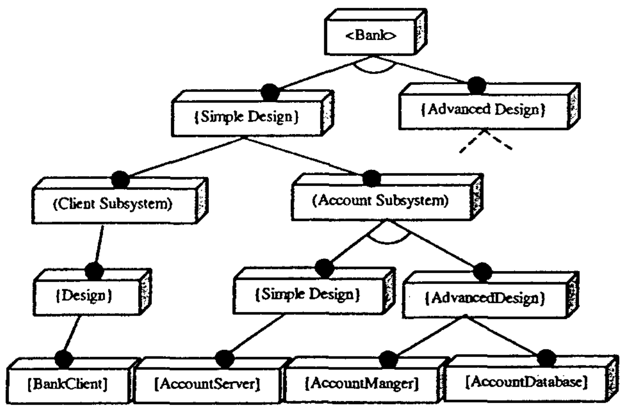


Figure 2. UA DSM for an account management system

Figure 2 shows a simplified example of a DSM for an account management system. In this DSM, two kinds of feature notations are used: mandatory and alternative. A node is mandatory if a simple edge ends with a filled circle touching it. This means this node is included in an architecture instance if and only if its parent is included. A set of nodes that is pointed to by edges connected by an arc forms alternatives. This means that if the parent of this set of nodes is included in an architecture instance, then exactly one node from this set is included in the architecture instance. The details of feature notations are indicated in [7]. The root of the example DSM is <Bank>, which indicates the specific type of account management system being considered. It has two different designs: a {Simple Design} and an {Advanced Design}. The details of the {Advanced Design} are omitted in the figure for simplicity. The {Simple Design} of the <Bank> has two subsystems: the (Client Subsystem) and the (Account Subsystem). These subsystems also can have more than one design that have different kinds of abstract components as shown in the Figure 2. Thus, this architecture can be configured, based on a customer's requirements, to create an appropriate architecture instance. One example of the customized architecture instance of this DSM is shown in Figure 3. Both the DSM and the architecture instance serve as the example in Section 4.

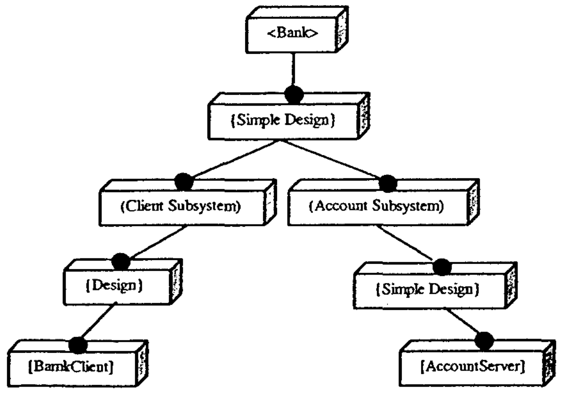


Figure 3. Architecture instance for an account management system

3.3 Language for ordering a DCS

Another important aspect of system level generative programming is how to express the query to order a concrete system out of a system family. [7] discusses the use of a domain specific language (DSL), which is a specialized and problem-oriented language, for placing an order. DSL could be a separate textual language, such as SQL, or it could be in a graphical notation. In general, there is a need for several different DSLs to specify a complete application. This makes the "order" complex. In UA, the ordering of a concrete system can be expressed in a structured form of natural language and then processed into TLG with the help of the query processor. TLG allows queries over the GDM to be expressed in a natural language-like manner, which is consistent with the way in which UMM is expressed. An example of a query for ordering a DCS is presented in Section 4.

3.4 UA generator

UA generator is a tool for realizing system-level generative programming. This generator is for system generation instead of component code generation. The architecture of he UA generator is shown in Figure 4. It consists of three functional modules: a generative domain model knowledgebase (GDMKB) producer; a query processor (natural language parser), which is responsible for translating natural language like orders into system specifications using TLG [8]; and an application producer which is responsible for assembling a DCS from a DCS family based on the UA GDM. The application producer implements the processing logic of the GDM. In our design, we separate UA GDM from the processing logic of GDM. The merit of this approach is that as a GDM evolves, the only thing that needs to be updated and maintained is the GDMKB. A simple generator for prototyping purposes has been designed and implemented with the logic of a multi-tiered architecture: client tier (web browser, HTML pages), web tier (web server, JSP/Servlet), business tier (application server, generator logic) and database tier (UA GDMKB). Experiments are underway with this prototype. The initial results indicate a good promise in a semi-automatic construction of simple distributed systems.

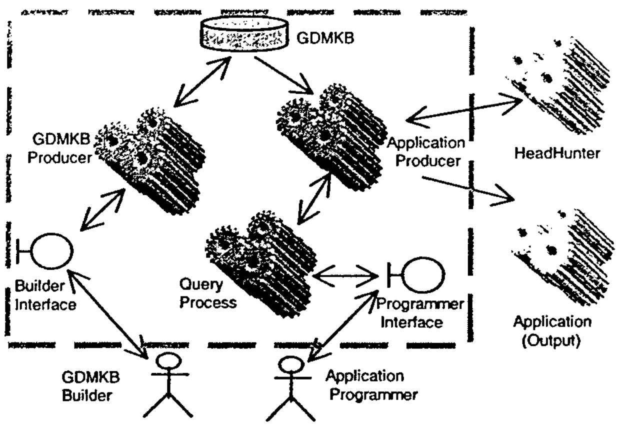


Figure 4. UA generator architecture

4. An example

In order to illustrate the process of the UA system-level generative programming, along with the functions of each of its constituents, a simple example of a bank account management system from the finance domain is described below. The DSM for this example is shown in Figure 2. This DSM constitutes four types of abstract components, [BankClient], [AccountServer], [AccountManager] and [AccountDatabase]. The goal is to assemble an account management system from the available concrete components of these abstract components using the corresponding generative domain model.

4.1 Determining the target system and its architecture instance

The general form of a query is to request the creation of a system that has certain QoS parameters. The name of the system is important in identifying the application domain. A sample query for the above example can be informally stated as: *Create a bank system for account management that has: end-to-end delay < 15 milliseconds and throughput > 2500 operations/second.* This query is parsed into a formal specification by the query processor. The generator checks the specifications against the GDM and may prompt for more information from the user, such as design option in this case (this is an iterative process to collect enough user requirements to determine the target system and its architecture instance).

Assume a simple design is specified for both the <Bank> and the (account subsystem) (certainly the UA generator provides the specifications from the GDM about the simple design and the advanced design so the application programmer can decide which one to choose). Then the generator can determine the architecture instance for the specified system and, thus, the required component types are also determined as seen in Figure 3. In this case, two types of component are needed to produce the desired system: [BankClient] and [AccountServer].

4.2 Searching for the concrete components

During this step, from the query and the available information in the DSM about the set of the required abstract components, searching criteria (for both functional and nonfunctional features) for each component type is created. In this example, the QoS of the two abstract components are set according to the QoS decomposition rules in this DSM: *1) component throughput > system throughput; 2) component end-to-end delay < system end-to-end delay.*

These decomposition rules provide the broadest range for the component QoS based on the system QoS. Thus the QoS criteria for both components are: a) throughput > 2500 operations/seconds; b) end-to-end delay < 15 milliseconds. Then the headhunters are contacted to search for the concrete components. If the found components are implemented with different technologies, the headhunters will also return the appropriate ICB. In this example, assume the headhunters discover the following concrete components for each of the required component types and the necessary ICB, Java-CORBA bridge (also a component type as described in section 2.3.1). Suppose these concrete components are implemented with different technologies: Java RMI or CORBA, and have different advertised QoS.

1. [BankClient]
   (a) BankClient - id: phoenix.cs.iupui.edu, technology: Java RMI, end-to-end delay < 10 milliseconds, throughput > 3000 operations/second
   (b) BankClient - id: lalo.cs.iupui.edu, technology: CORBA, end-to-end delay < 15 milliseconds, throughput > 2500 operations/second
2. [AccountServer]
   (a) AccountServer – id: swordmaster.cs.iupui.edu, technology: Java RMI, end-to-end delay < 5 milliseconds, throughput > 12000 operations/second
   (b) AccountServer – id: magellan.cs.iupui.edu, technology: CORBA, end-to-end delay < 1 millisecond, throughput > 8000 operations/second
3. [Java-CORBA bridge]
   Java-CORBA bridge – id: ericsson.cs.iupui.edu, technology: Java RMI, end-to-end delay < 1 millisecond, throughput > 10000 operations/second

4.3 System assembling and testing

Now the generator can assemble four systems (BankClient – AccountServer) from components found above. These four systems are distinguished by the implementation technology of its constituent components: Java RMI – Java RMI system, Java RMI – CORBA system, CORBA – Java RMI system and CORBA – CORBA system. The system QoS is composed from the QoS of the concrete components. The system QoS is used to select the final product. Assume the following composition rules for this example: *1) system throughput = min (component throughput); 2) system end-to-end delay = $\sum$component end-to-end delay.*

The system QoS of the four possible systems are listed below.

1. Java RMI – Java RMI system QoS
   system end-to-end delay < 15 milliseconds
   system throughput > 3000 operations/second
2. Java RMI – CORBA system QoS
   (including the Java-CORBA bridge)
   system end-to-end delay < 12 milliseconds
   system throughput > 3000 operations/second
3. CORBA – Java RMI system QoS
   (including the Java-CORBA bridge)
   system end-to-end delay < 21 milliseconds
   system throughput > 2500 operations/second
4. CORBA – CORBA system QoS
   system end-to-end delay < 16 milliseconds
   system throughput > 2500 operations/second

Based on the query and the analysis above according to the QoS composition rules, it is obvious the second system (Java RMI – CORBA) is the best. The first one (Java RMI – Java RMI) also meets the QOS requirement of the query. At this moment, the systems are chosen according to the advertised QoS of each component by QoS composition rules. The systems are further verified by the event grammars [2]. During system assembly, the code for carrying out event trace computations according to user-supplied test cases is also assembled. These test cases will be executed to verify that the assembled account management system does satisfy the system QoS specified in the query. If it does not, it is discarded. This verification process is carried out for each of the generated account management systems (the first two in the above example). The one with the actual best system QoS is chosen. If none of the systems meet the QoS criteria (as observed by an experimental evaluation), then the user may choose to modify the query and repeat the entire search, assembly and verification process.

5. Conclusion

The software solutions for the future DCS will require automatic or semi-automatic integration of software components, while abiding by the QoS constraints advertised by each component and the requirements on the system of components. This paper describes the system-level generative programming of the UA in the UniFrame that allows an effective and efficient assembly of heterogeneous and distributed software components to create a DCS out of a DCS family. The result of using the UniFrame and the associated tools (such as the UA generator) leads to the automation of DCS production while meeting both the functional and non-functional requirements of the DCS. Although a simple example is provided in this paper, the principles of the proposed approach are general enough to be applied to larger DCS.

6. Acknowledgement

This material is based upon work supported by the US Office of Naval Research under award number N00014-01-1-0746.

References

[1] Auguston, M. Program Behavior Model Based on Event Grammar and its Application for Debugging Automation. In *Proceedings of the 2nd Inernational Workshop on Automated and Algorithmic Debugging*, pages 277-291, 1995.

[2] Auguston, M., Gates. A., Lujan. M. Defining a Program Behavior Model for Dynamic Analyzers. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering. SEKE'97*, pages 257-262, 1997.

[3] Bryant, B.R. Object-Oriented Natural Language Requirements Specification. In *Proceedings of ACSC 2000, the 23rd Australasian Computer Science Conference, January 30-February 4, 2000, Canberra, Australia,* pages 24-30, January 2000.

[4] Bryant, B. R., Lee, B.-S. Two-Level Grammar as an Object-Oriented Requirements Specification Language, *Proceedings (CR-ROM) of 35th Hawaii International Conference on System Sciences, 2002,* page 10. http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDF documents/STDSLO1.pdf.

[5] Clements. P., Donohoe. P., Kang, K., Northrop, L. Fifth Product Line Practice Workshop Report. September, 2001. http://www.sei.cmu.edu/publications/documents/01.reports /01tr027.html.

[6] Cohen, S., Gallagher, B., Fisher, M., Jones, L., Krut, R., Northrop, L., O'Brien, W., Smith, D., Soule, A. Third DoD Product Line Practice Workshop Report. July 2000. http://www.sei.cmu.edu/publications/documents/00.reports /00tr024.html.

[7] Czarnecki, K., Eisenecker, U.W. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[8] Lee, B.-S., Bryant, B. R. Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language. *Proceedings of SAC 2002, the 2002 ACM Symposium on Applied Computing, March 11-14, 2002, Madrid, Spain, 2002*, pp. 932-936.

[9] Luqi, V. Berzins, J. Ge, M. Shing, M. Auguston, B.R. Bryant and B.K. Kin. DCAPS - Architecture for Distributed Computer Aided Prototyping System. In *Proceedings of the 12th IEEE International Workshop on Rapid System prototyping, pp.103-109, June 25-27, 2001, Monterey Beach Resort, California, USA,* IEEE Computer Society Press, 2001.

[10] Microsoft Corporation. DCOM Specifications, URL: - http://www.microsoft.com/oledev/olecom, 1998.

[11] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999. http://www.omg.org/cgi-bin/doc?orbos/99-02-05.

[12] Object Management Group (OMG). Model Driven Architecture: A Technical Perspective. Technical Report, OMG Document No. ab/2001-02-01/04. February 2001. ftp://ftp.omg.org/pub/docs/ab/01-02-04.pdf.

[13] Orfali, R., and Harkey, D. Client/Server Programming with JAVA and CORBA. The second edition. John Wiley & Sons, Inc., 1998.

[14] Raje, R. R., Bryant, B. R., Auguston, M., Olson, A M., Burt, C. C. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. *Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration, Monterey, California, 2001,* pp: 109-119.

[15] Raje, R. R., Auguston, M., Bryant, B. R., Olson A. M., Burt, C. C. A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components. Submitted for publication to Concurrency and Computation, 2001.

[16] Raje R. R. UMM: Unified Meta-object Model. *Proceedings of 4th IEEE International Conference on Algorithms and Architecture for Parallel Processing, ICA3PP'2000, pp: 454-465,* Hong Kong, 2000.

[17] Seigel, J. CORBA Fundamentals and Programming. John Wiley & Sons, Inc., 1996.

[18] Siram, N. N. An Architecture for the UniFrame Resource Discovery Service. MS thesis. Indiana University Purdue University Indianapolis, 2002.

[19] Software Engineering Institute. The Product Line Approach Initiative. http://www.sei.cmu.edu/plp/plp_init.html.

[20] Software Engineering Institute. A Framework for Software Product Line Practice-Version 3.0. http://www.sei.cmu.edu/plp/framework.html.

[21] Sun, C., Raje, R. R., Olson, A. M., Auguston, M., Bryant, B. R., Burt, C. C., Huang, Z. Composition and Decomposition of Quality of Service Parameters in Distributed Component-based Systems. To appear in *Prodeedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2002).*

Composition and Decomposition of Quality of Service Parameters in Distributed Component-Based Systems

Changlin Sun, Rajeev R. Raje,
Andrew M. Olson
Computer and Information Science
Indiana University Purdue University Indianapolis
723 W. Michigan Street, SL 280
Indianapolis, IN 46202, USA
{csun, rraje, aolson, csun}@cs.iupui.edu,
+1 317 274 5246/5174/9733

Mikhail Auguston
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
mikau@cs.nmsu.edu
+1 505 646 5286

Barrett R. Bryant, Carol Burt
Computer and Information Sciences
The University of Alabama at Birmingham
1300 University Blvd.
Birmingham, Alabama 35294-1170, USA
{bryant, cburt}@cis.uab.edu, +1 205 934 2213

Zhisheng Huang
Computer and Information Science
Indiana University Purdue University Indianapolis
723 W. Michigan Street, SL 280
Indianapolis, IN 46202, USA
zhuang@cs.iupui.edu
+1 317 274 5246

Abstract

It is becoming increasingly acceptable that the component-based development is an effective, efficient and promising approach to develop distributed systems. With components as the building blcks, it is expected that the quality of the end system can be predicted based on the qualities of components in the system. UniFrame is one such framework that facilitates a seamless interoperation of heterogeneous distributed software components. As a part of UniFrame, a catalog of quality of service (QoS) parameters has been created to provide a standard method for quantifying the QoS of software components. In this paper, an approach for composition and decomposition of these QoS parameters is proposed. A case study from the financial domain is indicated to validate this model.

Keywords: Distributed systems, software components, quality of service (QoS), composition, decomposition

1. Introduction

The development of distributed systems from reusable components is becoming increasing important because of its potential to reduce product development cost and time-to-market. Unfortunately, the current component-based approaches concentrate mainly on functional properties, and ignore the quality of service (non-functional) properties, which are crucial in many application domains. A few examples of quality of service parameters are: *dependability, reliability, availability, maintainability, adaptability, portability, evolvability, achievability, security, presentation, throughput, result, and turnaround time* [3]. In a component-based approach, it is relatively easy to glue components together to provide the desired system functionality, but it is difficult to guarantee the quality of service provided by a system made up of individual components. Hence, it is critical to determine the distribution of a system property into its component properties (decomposition) and how to reason the system property from the property of its individual components (composition). Currently, there is no common and accepted design standard that can facilitate such composition and decomposition.

In [1, 2] a framework, UniFrame, based on a unified meta-component model (UMM) and a unified approach (UA), is proposed for building distributed component-based systems. In UMM, components are autonomous entities that provide services and guarantee their quality. The creation of a distributed software system using UA has two levels: a) component level - developers create components, test and validate the appropriate QoS and deploy the components on the network, and b) system level – a collection of components, each with a specific functionality and QoS, enables a semi-automatic generation of a distributed software system. The focus of this paper is to study the mechanism of decomposition and composition of various QoS parameters so that the properties of the entire system can be inferred from the QoS properties of individual parameters and vice versa. The first step towards composition and decomposition is to identify and classify various QoS parameters.

2. Classification of QoS parameters

In [3], sixteen QoS parameters are identified and described in a catalog. The aim of this section is to study these parameters from the perspective of composition and decomposition, and classify them into different categories. Such a classification provides the developer of distributed systems the knowledge about how these QoS parameters should be treated during the creation of a software realization of a distributed system.

2.1 Static and dynamic QoS parameters

Static QoS parameters can be evaluated by examining the internal structure of a software component. These parameters are stable in different environments provided the internal structure of component is unchanged. The examples of static QoS parameters are *reliability, maintainability, portability, scalability, reusability, presentation, usability, security, priority, and parallelism constraints*. Dynamic QoS parameters, on the other hand, can be measured by observing the system behavior at run-time. These parameters are tightly associated with the deployment environment. Examples of dynamic parameters are *throughput, turnaround time, capacity, availability, and result*.

Static QoS parameters may compose well as they do not tend to change during system execution. However, the execution environment, which is not known in advance, influences dynamic QoS parameters and makes their composition a difficult task.

2.2 Application dependent and independent QoS parameters

Different application domains require the use of different QoS parameters. For example, in the E-commerce applications, availability, turnaround time, throughput and usability are important, while in the visualization applications, the frame rate is critcal. Some parameters are application dependent (e.g., throughput), while some others are application independent (e.g., reusability). Obviously, the application independent parameters are more convenient to deal with than the application dependent parameters, because the latter need application-specific information.

2.3 Parameters with different ranges of decomposition

The dependence of a system level property on the component level property leads to several special decompositions of QoS parameters: *universal, subset, existential* and *component-specific* decomposition. For universal decomposition of QoS parameters, the system level property decomposes into all of the components in the system. Most of the QoS parameters have universal decomposition, such as, availability, reliability, security, etc. For subset decomposition of QoS parameters, the system level property decomposes into a subset of components in the system. For existential decomposition of QoS parameters, the system level property decomposes into any component in the system. Mobility is an example of QoS parameters with existential decomposition. For component-specific decomposition of QoS parameters, the system level property decomposes into a particular component. For example, presentation of a system is decomposed into the presentation provided by the user-interface component of the system.

2.4 Parameters with different aggregation rules

In the physical world, some properties show different aggregation rules. For example, the mass or the energy of a system is the sum of the mass or the energy of subsystems. The density or the temperature of a system is the average of the density or the temperature of subsystems. The strength of a system is limited by the strength of the subsystem with the minimum value of strength. Similarly, for systems built from software components, different QoS parameters may abide by different composition rules. For example, the turn-around time of a system is the sum of the turn-around time of each component in the system. The maintainability of a system is an average of the maintainability of each component in the system. The security of a system is limited by the component with the minimum value of security.

3. System decomposition and composition models

In this section, a decomposition and composition model of QoS parameters is proposed. The model includes the decomposition process, the composition process, and the corresponding rules.

3.1 Decomposition rules

The decomposition process factorizes the system QoS parameters to QoS parameters of components and provides a rough estimate of the values for the QoS parameters of individual components. To decompose the QoS parameters of systems, we identify following properties: *at-least-one property, universal property, subset property, at most one property, at-least-one-X property, universal-X property, subset-X property* and *at-most-one-X property.*

A property X is an at-least-one property if, when any system has property X, at least one component of that system has property X. A property X is a universal property if, when any system has property X, all components of that system have property X. A property X is a subset property if, when any system has property X, a subset of components of that system have property X. A property X is an at-most-one property if, when any system has property X, at most one component of that system has property X. A property Y is an at-least-one-X property if, when any system has property Y, at least one component of that system has a certain property called X. A property Y is a universal-X property if, when any system has property Y, all components of that system have a certain property called X. A property Y is a subset-X property if, when any system has property Y, a subset of components of that system have a certain property called X. A property Y is an at-most-one-X property if, when any system has property Y, at most one component of that system has a certain property called X.

Each QoS parameter needs to be classified using one of these properties. For example, from the decomposition point of view, mobility is an at-least-one property, security is a universal property, and frame rate is a universal-X property, where X is throughput. Based on the definitions of these properties, the QoS parameters for individual components can be classified into one of these properties. Given the value of system-level QoS parameters, an upper or lower bound of the value of the QoS parameters of an individual component can be estimated. For example, for turn-around time, the component turn-around time TAT_i (i=1, 2, ..., n) need to satisfy $0 < TAT_i < TAT$, where TAT is the system turn-around time, while for security, the component security S_i (i=1, 2, ..., n) need to satisfy $S_i > S$, where S is the system security requirement.

3.2 Composition rules

During the composition process, the system QoS parameter is reasoned from the QoS parameters of components. Due to the causal link between the property of the system and the properties of components in the system, we assume the property of composed system depends on the properties of components in the system. The proposed equation for composition can be written as:

$$P = f(p_1, p_2, \ldots, p_n) \qquad (1)$$

where P is the property of composed system, and p_i (i=1.2, ...,n) is the property of component i in the system.

The equation (1) can be approximated as a weighted sum, as indicated below:

$$P = w_1 \cdot p_1 + w_2 \cdot p_2 + \ldots, + w_n \cdot p_n \qquad (2)$$

Where w_i (i=1, 2, ..., n) is a constant coefficient (weight), for the component I, within the range [0, 1]. The determination of w_i is based both on the analysis and experimentation.

For each QoS parameter, the equation (2) can be simplified. For example, in the case of maintainability, equation (2) becomes:

$$P = w_1 \cdot p_1 + w_2 \cdot p_2 + \ldots, + w_n \cdot p_n \qquad (3)$$

where $w_i = \frac{LOC_i}{\sum_{j=1}^{n} LOC_j}$, LOC=Lines of Code.

For QoS parameters: security, adaptability, capacity, equation (2) becomes:

$$P = Min\,(p_1, p_2, \ldots, p_n) \qquad (4)$$

For the QoS parameter turn-around time, equation (2) becomes:

$$P = p_1 + p_2 +, \ldots, + p_n \qquad (5)$$

Theoretically, for each QoS parameter, a corresponding composition rule can be derived from equation (2) based on analysis and experimentation.

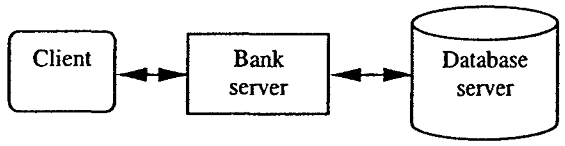


Figure 1. Bank account system architecture

4. A case study

To illustrate the composition and decomposition of QoS parameters in developing distributed component-based system, a simple bank account system (financial system) is discussed below. As shown in Figure 1, the system consists of three components: client, bank server, and database server. For this bank account system, the following QoS parameters, based on the functionality, are identified: availability, turnaround time, security, throughput, reliability, and usability. As turn-around time is an important dynamic QoS parameter for most applications, its composition rule is validated through experiments as indicated below.

The experimental system, as shown in Figure 2, consists of three Ultra-250, SPARC Sun workstations. The workstations Phoenix and Magellan are connected using a 100Mbit Ethernet and the workstation Raleigh is connected via a 10Mbit Ethernet. Phoenix is a file server for the local area network.

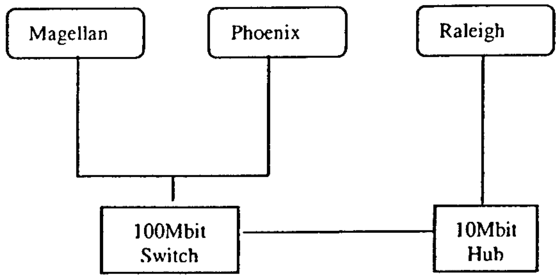


Figure 2. Experimental setup

The three components in the bank account system are implemented using Java RMI. For the purpose of the validation, each component has an instrumented code to measure the dynamic QoS parameters during the execution time.

Initially, the turnaround time for each component is measured by running them (in isolation) on each of the three workstations multiple times. These experiments yielded the following average turnaround times for the three components under consideration: 34 ms (client component), 119 ms (bank server component) and 126 ms (database server component). Based on the composition rule for turn-around time, the predicted turn-around time for the entire system is the summation of the individual turnaround times, i.e., the turnaround time for the system is predicted to be 278 ms. To experimentally validate this predicated value, the three components were deployed using two distributed configurations: a) the client on Raleigh, bank server on Phoenix and database server ob Magellan, and b) the client on Raleigh and both the servers on Magellan. In both the cases, the system level turn-around time was measured. The error between the predicted turnaround time and the actual turnaround time, for both the configurations, was found to be of 3.3% and 3.1% respectively. Hence, it can be concluded from these simple experiments that the model presented here allows the prediction of values for the turnaround time with a good accuracy. Similar empirical studies for validating the composition rules for other parameters are being carried out. However, for the sake of brevity, and to adhere to the space constraints, these are not reported in this paper.

5. Conclusions

The UniFrame approach provides a framework for the development of distributed software systems based on components by highlighting not only the functional but also the QoS requirements. The QoS feature of a system can be predicted by applying the composition and decomposition rules to QoS attributes of the individual components. These rules are based on the classification of different QoS parameters. A simple case study presented here empirically validates the composition/decomposition rules described in this paper.

6. Acknowledgement

This material is based upon work supported by the US Office of Naval Research under award number N00014-01-1-0746.

References

[1] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson and C. Burt. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. Proceedings of the 2001 Monterey Workshop, Monterey, California, June 2001, pp. 109-119.

[2] R. Raje, M. Auguston, B. Bryant, A. Olson, C. Burt. A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components. Technical Report, Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2001.

[3] G. Brahnmath, R. Raje, A. Olson, M. Auguston, B.Bryant, C. Burt. A Quality of Service Catalog for Software Components. Proceedings of the Southeastern Software Engineering Conference, Huntsville, Alabama, April 2002.

Automation of Software System Development Using Natural Language Processing and Two-Level Grammar

Beum-Seuk Lee and Barrett R. Bryant

Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL 35294-1170 U. S. A.
{leebs, bryant}@cis.uab.edu

Abstract. In software engineering, even with recent active research on formal methods and automated tools, users' involvement is inevitable and crucial throughout the software development lifecycle. Automation of these manual tasks would assist the developers throughout the development. Our project goal is to help the engineers to resolve ambiguity in natural language (NL) using Natural Language Processing and to overcome different levels of abstraction between requirements documents and formal specifications using Two-Level Grammar (TLG). The result is a system that assists developers to build a formal representation from the informal requirements for rapid prototyping and even implementation.

Keywords: Natural Language Processing, Formal Specification, Automated Software Engineering, Two-Level Grammar (TLG)

1 Problem Statement

Even the rigorous development of formal specifications and automated tool kits in recent years hasn't eliminated the practical importance of requirements documents written in natural language and the necessity of users' involvement throughout the software development life cycle.

Even though natural language is inherently object-oriented and descriptive with strong representation power, its syntax and semantics are not formal enough to be used directly as a programming language. Therefore the requirements documentation written in NL has to be reinterpreted into a formal specification language by software engineers. Pohl rightly stated regarding this process that improving an opaque system comprehension into a complete system specification and transforming informal knowledge into formal representations are the major tasks in requirements engineering [1]. When the system is very complicated, which is mostly the case when one chooses to use formal specification, this conversion, if manually done, is both non-trivial and error-prone, if not implausible.

Many similar tasks of manual involvement occur and are repeated to translate the requirements documents into a formal specification or into final executable code regardless the type of the system under development. Some examples of these tasks are domain-specific knowledge collection, correct interpretation of requirements, specification update, and maintenance of consistency, to name a few.

It is well known that as much as 60 percent of the errors that appear during a system's life cycle have their origin in the requirements phase [2]. It is also well known that the closer to correct an error found in the development and later stages of system development is orders of magnitude higher than to correct the same error found during the requirements stage [3]. Therefore ensuring the correctness of the requirements as well as their interpretation and translation cannot be overemphasized.

The challenge of formalizing a natural language requirements document, which takes up major portion of human involvement in the system development, results from many factors such as miscommunication between domain experts and engineers. However the major bottleneck of this conversion is from the inborn characteristic of ambiguity of NL and the different level of the formalism between the two domains of NL and formal specification.

To handle this ambiguity problem, some have argued that the requirements document has to be written in a particular way to reduce ambiguity in the document [4]. Others have proposed controlled natural languages (e.g., Attempto Controlled English (ACE) [5]) which limit the syntax and semantics of NL to avoid the ambiguity problem. Another approach to NL requirements analysis is to search each line of the requirements document for specific words and phrases for the purpose of quality analysis [6]. A similar project [7] focuses mainly on the automatic indexing and reuse of the software components in the requirement documents. However there has been no attempt to automate the conversion from requirements documentation into a formal specification language for prototyping as well as implementation.

In our research, Natural Language Processing (NLP) [8] is used to handle the ambiguity problem in NL and Two Level Grammar (TLG) [9] is used to deal with the different formalism level between NL and formal specification languages to achieve the automated conversion from NL requirements documentation into a formal specification (in our case VDM++ [10] - an object-oriented extension of the Vienna Development Method [11]) and to reduce and reuse the developers involvement.

2 Introduction

To achieve the conversion from requirements documents to a formal specification several levels of conversions are required. First the original requirements written in natural language is to be refined as a preprocessing of the actual conversion. This refinement task involves checking spellings, grammatical errors, consistent use of vocabularies, organizing the sentences into the appropriate sections, etc.

Next the refined requirements document is expressed in XML format. By using XML to specify the requirements, XML attributes (meta-data) can be added to the requirements to interpret the role of each group of the sentences during the conversion. The information of the domain-specific knowledge is specified in XML. The domain-specific knowledge describes the relationship between components and other constraints that are presumed in requirements documents or too implicit to be extracted directly from the original documents.

Then a knowledge base is built from the requirements document in XML using NLP to parse the documentation and to store the syntax, semantics, and pragmatics information. In this phase, the ambiguity is detected and resolved, if possible. Once the knowledge base is constructed, its content can be queried in NL. Next the knowledge base is converted, with the information of the domain specific knowledge, into Two Level Grammar (TLG) by removing the contextual dependency in the knowledge base. TLG, the most NL-like specification language which is a unification of functional, logic, and object-oriented programming styles, is used as an intermediate representation to build a bridge between the informal knowledge base and the formal VDM++ representation.

Finally the TLG code is translated into VDM++ by data and function mappings. VDM++ is chosen as the target specification language because VDM++ has many similarities in structure to TLG and also has a good collection of tools for analysis and code generation. Once the VDM++ representation of the specification is acquired we can do prototyping of the specification using the VDM++ interpreter. Also we can convert this into a high level language such as Java^{TM} or C++ or into a model in the Unified Modeling Language (UML) [12] using the VDM++ Toolkit [13]. The entire system structure is shown in Figure 1.

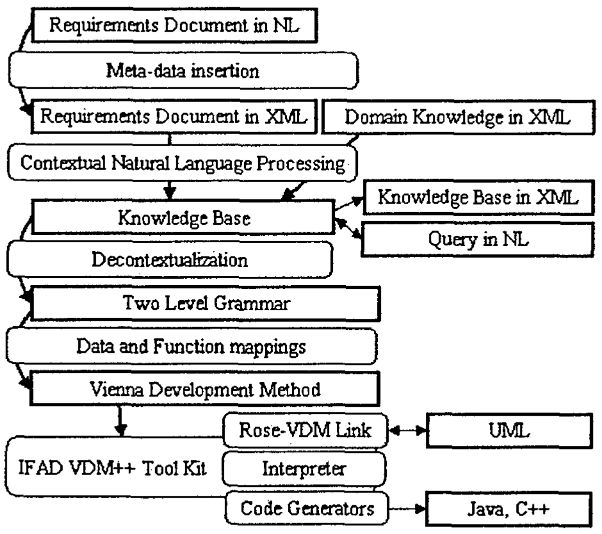


Fig. 1. System Structure.

The translation of our system is incremental and iterative reflecting the changes made throughout the system development. The user interaction is likely to happen at any stage of the translation to supervise and assist the automation. By keeping track of user's preferences and configurations for each iteration and automating the translations accordingly, the user's involvement can be reasonably reduced.

In the sections which follow, we will present the following simplified (thus incomplete) Computer Assisted Resuscitation Algorithm (CARA) [14] Infusion Pump Control System to illustrate our approach and describe the various system components.

```
HOST is powered up and all software subsystems are available.
The pump software system is now in the wait operating state. Patient
with IV/pump running is placed onto the HOST. Pump cable is connected
to the HOST. HOST now provides power for pump. Pump software system
detects pump connection and monitors occlusion and airlock logic levels.
Pump subsystem display is automatically brought forward to the secondary
display. Pump software subsystem detects back EMF and fluid impedance
and begins to log infusion rate. Pump continues to operate on it's
hardware setting. Pump software system is now in manual operating state.
One of the blood pressure sensors is connected to the patient.
Pump software system detects clean blood pressure signal and activates
automatic servo-control start button. When the start button is pressed
the MAC controls the pump and begins resuscitation to the prescribed
blood pressure setpoint. The system is now in the automatic
servo-control on operating state when the pump is infusing fluid into
a patient using the hardware (HW) flow setting on the pump. If for any
reason (change IV bags, change or fix blood pressure sensor, etc.) it
becomes necessary to pause the MAC, the pause button on the display may
be pressed. This causes the infusion pumping to cease. The system is
now in the automatic servo-control paused operating state. The system
maybe restarted at any time. When the patient is to be removed from
the HOST, the pump software system should be returned to the manual
operating state. The blood pressure sensor should be removed from the
patient and then the pump cable can be removed from the HOST.
This allows the pump to continue operating in standalone mode or the
IV infusion to be discontinued.
```

3 Requirements in XML

Rearranging related information together in the requirements will ease the conversion. Specially because we are assuming that the requirements can contain different aspects of information (functional, non-functional or even a mixture of both) about the system. Even requirements that are functionality-oriented can have different types of functionality. For example, they can be object-oriented, procedural, real time-based, event-based, etc. Rearranging related information together will ease the conversion. This can be achieved by specifying the role of each paragraph using XML data structure and notations. This will help the knowledge base to maintain the correct structure.

The CARA specification in XML is shown as follows.

```
<document>
<c title = "Mode" meta = "mode">
 <c title = "wait state" meta = "submode">
  <p meta = "pre_cond">
   <s>HOST is powered up and all software subsystems are available</s>
  </p>
  <p meta = "pre_exec">
   <s>Patient with IV/pump running is placed onto the HOST</s>
   <s>Pump cable is connected to the HOST</s>
  </p>
  <p meta = "exec">
   <s>HOST now provides power for pump</s>
  </p>
  <p meta = "break_cond">
   <s>When the pump is infusing fluid into a patient using
  the hardware (HW) flow setting on the pump the system is no longer in
  the wait state</s>
  </p>
 </c>
 <c title = "manual state" meta = "submode">
  <p meta = "pre_exec">
   <s>Pump software system detects pump connection and monitors occlusion
    and airlock logic levels </s>
   <s>Pump subsystem display is automatically brought forward to the
  secondary display</s>
   <s>Pump software subsystem detects back EMF and fluid impedance and
  begins to log infusion rate</s>
   <s>Pump continues to operate on it's hardware setting</s>
   <s>One of the blood pressure sensors is connected to the patient</s>
   <s>Pump software system detects clean blood pressure signal and
  activates automatic servo-control start button</s>
  </p>
 </c>
 <c title = "autocontrol on state" meta = "submode">
  <p meta = "pre_exec">
   <s>When the start button is pressed the MAC controls the pump and
  begins resuscitation to the prescribed blood pressure setpoint</s>
  </p>
 </c>
 <c title = "autocontrol paused state" meta = "submode">
  <p meta = "pre_exec">
   <s>If for any reason (change IV bags, change or fix blood pressure
  sensor, etc.) it becomes    necessary to pause the MAC, the pause
  button on the display may be pressed</s>
   <s>This causes the infusion pumping to cease</s>
  </p>
  <p meta = "break_cond">
   <s>The system maybe restarted at any time</s>
  </p>
  <p meta = "break_exec">
   <s>When the patient is to be removed from the HOST, the pump software
  system should be returned to the manual operating state</s>
   <s>The blood pressure sensor should be removed from the patient and
  then the pump cable can be removed from the HOST</s>
   <s>This allows the pump to continue operating in standalone mode or
  the IV infusion to be discontinued</s>
  </p>
 </c>
</c>
</document>
```

The meta attribute in XML indicates the role of each paragraph. Namely it shows if the group of the sentences describes state types (mode), execution types (_exec), various conditions (_cond), etc. submode indicates the state. In the CARA example, there are four distinctive states; wait state, manual state, autocontrol on state, and autocontrol paused state. In a state, preconditions (pre_cond) have to be satisfied to enter the state. Some statements (pre_exec) will be executed when the system enters into a state. Other statements (exec) will be executed while the system is in the state. If any break conditions (break_cond) are satisfied in the state, the system will leave the state. There may be some cases where break conditions will execute some statements (break_exec) before breaking out of the state. Also some default statements (post_exec) are executed before leaving the state. We have specified these meta attributes for various types of functionality in requirements to cover a wide range of different requirements documents. Using a tree-like structure in XML the specifications become more descriptive as the tree expands further. Organizing and representing the requirements document in XML according to the roles of the specifications of the system not only enhances understanding of specifications but also helps to standardize requirements composition.

4 Domain-Specific Knowledge in XML

A requirements document usually contains specific information about how the system should work whereas the domain knowledge describes how the system is composed by its components and the constraints imposed on the components or on the relations among them. The domain-specific knowledge is a world knowledge specific to a certain domain in which the system is defined. This is well tied into the concept of the family or the ontology of systems. Depending on the level of abstraction (or the details described) of the domain knowledge, the effort to construct it can vary. By limiting the level of abstraction, the body of the knowledge can be reduced into a reasonable size and so can the effort to build it. Usually the domain-specific knowledge is defined informally or only for a specific project, not reusable or extensible for similar systems (the systems in the same family). By using XML to specify the domain knowledge with a minimum semantics, not only can the specification be formally defined but also it can be extensible gradually building up an ontology of systems.

In our research the domain knowledge specified in XML shares many similarities with DARPA Agent Markup Language (DAML) [15] which is a frame-based language with semantics to describe ontology. Because domain knowledge is more than just an ontology, DAML is not expressive enough to describe the whole aspect of the domain knowledge. However using the XML syntax a domain knowledge can be specified in various ways leaving the interpretation of its semantics totally up to the system that uses it [16]. Therefore when a specification for domain-specific knowledge in XML is to be developed, its formal semantics as well as its expressiveness has to be considered at the same time.

The following describes an example of the domain knowledge of `Car` to illustrate the use of domain-specific knowledge expressed in XML in our project.

```
<system name = "Car">
 <component name = "Engine">
  <amount type ="exactly" value = "1"/>
  <unit type = "volume" value = "liter"/>
  <subcomponent name="Cylinder" type = "integer">
   <amount type ="one_of" value = "4,6,8"/>
  </subcomponent>
  <relation with = "Starter" type ="pass_to" value ="signal"/>
 </component>
 <component name = "Wheel"/>
 <component name = "Body">
  <relation with = "Frame" type ="synonym"/>
 </component>
 <relation with = "Vehicle" type ="inheritance" value ="parent"/>
 <relation with = "Van" type ="inheritance" value ="child"/>
</system>
```

According to the above domain specification, `Car` is composed of `Engine`, `Wheel`, `Control`, and `Body`. `Vehicle` is a parent of `car` whereas `Van` is a type of `Car`. `Car` can have exactly one `Engine` and the unit of `engine` is volume expressed in liters. `Engine` has `Cylinder` as its subcomponent. The number of `Cylinders`, which is as an integer number and is representative part of the subcomponent, can be either 2, 6, or 8. `Starter` passes a signal to `Engine` (to turn the motor). `Body` of `Car` also can be called as `Frame`.

The following is Document Type Definition (DTD) for the domain knowledge in XML, which defines the formal semantics of the domain-specific knowledge while pertaining proper expressive power.

```
<!ELEMENT system (component|relation)*>
<!ELEMENT component (amount?, unit?, (subcomponent|relation)*)>
<!ELEMENT subcomponent amount?>
<!ELEMENT amount EMPTY>
```

```
<!ELEMENT unit EMPTY>
<!ELEMENT relation EMPTY>

<!ATTLIST system name CDATA #REQUIRED>
<!ATTLIST component name CDATA #REQUIRED>
<!ATTLIST subcomponent name CDATA #REQUIRED type CDATA #IMPLIED>
<!ATTLIST amount type CDATA "exactly" value CDATA #REQUIRED>
<!ATTLIST unit name CDATA #IMPLIED type CDATA #REQUIRED>
<!ATTLIST relation with CDATA #IMPLIED type CDATA #REQUIRED value
 CDATA #IMPLIED>
```

Note that the domain-specific knowledge in XML for the translation doesn't have to describe the domain exhaustively. Namely most of the elements and attributes are optional and attribute values can be any character strings. For example, the relationship element can represent inheritance, acronyms, message passing, etc. The minimum information required to guide the translation would be sufficient with the possibility of adding on more information later when necessary.

The domain knowledge for our CARA example is shown as follows.

```
<system name = "CARA system">
 <component name = "Computer Assisted Resuscitation Algorithm">
  <subcomponent name = "Display"/>
  <subcomponent name = "Button"/>
  <subcomponent name = "Pump Software System"/>
  <subcomponent name = "MAC"/>
  <relation with = "System" type = "hypernym"/>
  <relation with = "Software" type = "hypernym"/>
  <relation with = "Algorithm" type = "hypernym"/>
  <relation with = "CARA" type = "acronym"/>
 </component>
 <component name = "Patient"/>
 <component name = "HOST">
   <subcomponent name = "Pump"/>
 </component>
</system>
```

The above specification describes that the whole system is composed of by `Computer Assisted Resuscitation Algorithm`, `Patient`, and `HOST`. `Computer Assisted Resuscitation Algorithm` is a type of `Algorithm`, `Software`, or `System` that can be abbreviated as `CARA`.

In the natural language documents one concept can be represented by many different ways causing the translation hard to cluster similar information together. These can be acronym, synonym, and hypernym. From the CARA example, the word Computer Assisted Resuscitation Algorithm' is interchangeable with 'Algorithm' or 'CARA'. By using a minimum set of representative words that describes the entire components in the domain-specific knowledge, one-to-many relations between words and their various representations can be obtained and thus provides a simpler source to translate. The full set of words in the requirements documents are mapped into the minimum set of representative words by measuring similarity among words. The hypernym and the location of the common words are used for this estimation.

In summary, by specifying domain-specific knowledge in XML and limiting the scope of the knowledge the effort needed to build up the domain knowledge for the translation can be greatly reduced.

5 Conversion from XML to Knowledge Base

The raw information of the requirements document in natural language is not in the proper form to be used directly because of the ambiguity and implicit semantics in the document. Therefore an explicit and declarative representation (knowledge base) is needed to represent, maintain, and manipulate knowledge about a system domain [17]. Not only does the knowledge base have to be expressive enough to capture all the critical information but also it has to be precise enough to clarify the meaning of each knowledge entity (sentence). In addition, the knowledge base has to reflect the structure of TLG into which the knowledge base is translated later.

The knowledge base isn't a simple list of sentences in the requirements document. The linguistic information of each sentence such as lexical, syntactic, semantic, and most importantly discourse level information has to be stored with proper systematic structure.

Each sentence of the requirements documents has to be represented in a way that eases the interpretation of the sentence. In computational linguistics this is done by constructing a parse tree of the sentence, which contains the syntactic information of the sentence. By using this semantic information we can tell what type of operation a certain object executes on other objects.

To build a parse tree, each sentence in the requirements document is read by the system and tokenized into words. At the syntactical level, the part of speech (e.g. noun, verb, adjective) and the part of sentence (e.g. subject and object) of each word are determined by standard parsing techniques [8]. The corpora of statistically ordered parts of speech (frequently used ones being listed first) of about 85,000 words from Moby Part-of-Speech II [18] are used to resolve the syntactic ambiguity when there is more than one valid parsing tree. The system is able to handle elliptical compound phrases, comparative phrases, compound nouns, and relative phrases to allow the natural language in the requirements documents to be less controlled thus more natural.

Also the anaphoric references (pronouns) in a sentence are identified according to the current context history. A pronoun can represent a word, sentence, or even context. It is worthwhile to mention here that the requirements documents are easier to process than other types of textual documents in the sense that usually requirements documents have well defined structures with less ambiguities and infrequent use or narrow reference scope of pronouns.

Once the references of pronouns are determined, each sentence is stored into the proper context in the knowledge base. The structure of the knowledge base reflects the structure of the requirements in XML. The meta attribute information from XML is also stored in the knowledge base to be used for the translation from knowledge base into TLG. If no meta attribute or data structure is specified in the requirements in XML, the system totally relies on the linguistic information in the document to build the knowledge base according to the context. For more information on this process, we refer the readers to [19]. A part of the CARA knowledge base is shown in the Figure 2. The knowledge base of the

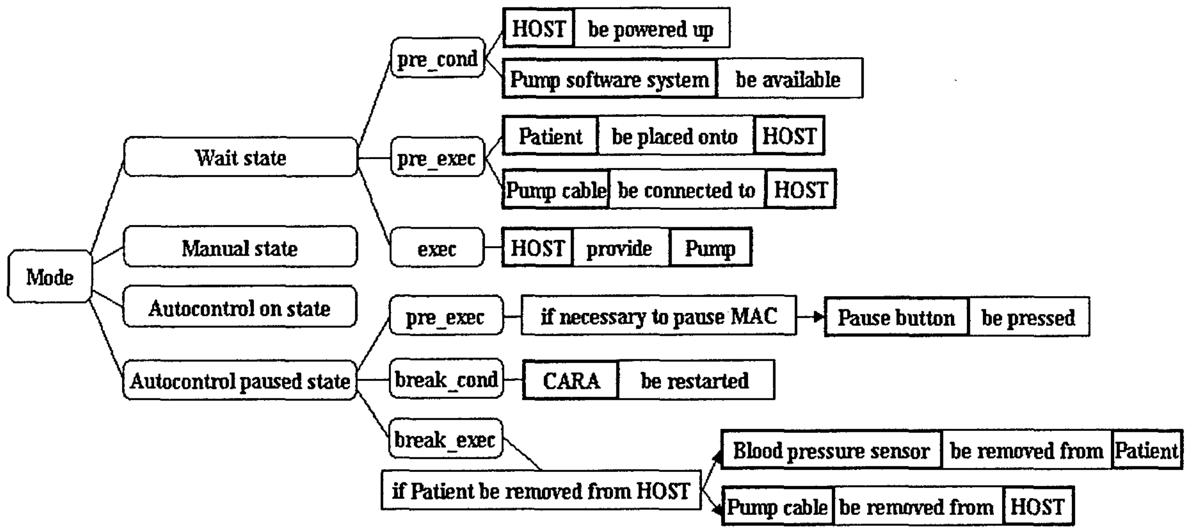


Fig. 2. Knowledge base for CARA.

CARA system contains the meta information from the XML requirements in its tree-like structure as well as the linguistic knowledge.

In summary, the knowledge base stores not only the linguistic information of each sentence but also the data structure and meta information of related sentences as specified in the requirements in XML. Along with this process, linguistic ambiguity is detected and resolved in parsing and construction of the knowledge base.

6 Transition from Knowledge Base to TLG

Two-Level Grammar (TLG) may be used to achieve translation from an informal NL specification into a formal specification. Even though TLG has NL-like syntax its notation is formal enough to allow formal specifications to be constructed using the notation. It is able not only to capture the abstraction of the requirements but also to preserve the detailed information for implementation. The term "two level" comes from the fact that a set of domains may be defined using context-free grammar, which may then be used as arguments in predicate functions defined using another grammar. TLG may be used to model any type of software specification. The basic functional/logic programming model of TLG is extended to include object-oriented programming features suitable for modern software specification [20]. The syntax of the object-oriented TLG is:

```
class Class_Name.
  Data_Name {, Data_Name}::Data_Type {, Data_Type}.
  Rule_Name : Rule_Body {, Rule_Body}.
end class [Class_Name].
```

where the term that is enclosed in the curly brackets is optional and can be repeated many times, as in Extended Backus-Naur Form (EBNF). The data types of TLG are fairly standard, including both scalar and structured types, as well as types defined by other class definitions. The rules are expressed in NL with the data types used as variables.

The conversion from the knowledge base to TLG flows very nicely because the knowledge base is built with the structure taking this translation into consideration. The root of each context tree becomes a class. And then the body of each class is built up with the sentence information in the sub-contexts of the root. Combined with the specification in the domain-specific knowledge, the knowledge base of the CARA example would be translated into the following TLG specification.

```
class Mode.

  main :
    wait state;
    manual state;
    autocontrol on state;
    autocontrol paused state.

  wait state:
    HOST is powered up,
    Pump_Software_System is available,
    Patient is placed onto HOST,
    Pump Cable is connected to HOST,
    while true then
      if Pump is infusing Fluid into Patient then
        break,
      HOST provide Power for Pump
    end while.

  manual state:
    Pump_Software_System detects Pump_Connection,
    Pump_Software_System monitors Occlusion and Airlock_Logic_Levels,
    Pump Display is brought to Secondary_Display,
    Pump_Software_System detects Back_EMF and Fluid_Impedance,
    Pump_Software_System begins to log Infusion_Rate,
    Pump continue to operates on Hardware_Setting,
    Blood_Pressure_Sensor is connected to Patient,
    Pump_Software_System detects Clean_Blood_Pressure_Signal,
    Pump_Software_System activates Automatic_Servo-control_Start_Button.

  autocontrol on state:
    if Start_Button is pressed then
      MAC controls Pump,
      MAC begins Resuscitation to Prescribed_Blood_Pressure_Setpoint
    end if.

  autocontrol paused state:
    if necessary to pause MAC then
      Pause_Button is pressed,
      cause Infusion_Pumping to cease
    end if,
    while true then
      if Patient is removed from HOST then
        Blood_Pressure_Sensor is removed from Patient,
        Pump Cable is removed from HOST,
        allow Pump to continue operating in Standalone_Mode,
        allow IV_Infusion to be discontinued,
        break
      end if
    end while.

end class.
```

The main function will execute all 4 state functions (`wait state`, `manual state`, `autocontrol on state`, `autocontrol paused state`) in parallel. However preconditions (pre_cond) in each state will be used as guarded statements to determine which state the system is currently in. For each state function, first the preconditions will be checked. If all the preconditions are met, `pre_exec` statements are executed once. Then in the infinite while loop `exec` statements are executed. If break_exec and break_cond statements are used for the system to break out the loop. If there are any `post_exec` statements, they are executed before returning from the function.

The TLG code is translated into VDM++ by data and function mappings (for more details on this translation we refer the readers to [9]). Once we have translated the TLG specification into a VDM++ specification we can convert this into a high level language such as Java^{TM} or C++, using the code generator that the VDM++ Toolkit^{TM} provides. Not only is this code quite efficient, but it may be executed, thereby allowing a proxy execution of the requirements. This allows for a rapid prototyping of the original requirements so that these may be refined further in future iterations. Namely the inconsistencies, contradictions, and ambiguities hidden in the informal description can be discovered in the formal representation using the VDM++ Toolkit. Another advantage of this approach is that the VDM++ Toolkit also provides for a translation into a model in the Unified Modeling Language (UML) using a link with Rational Rose^{TM}.

7 Contribution and Conclusion

This research project is developed as an application of formal specification and computational linguistic techniques to automate the conversion from a requirements document written in NL to a formal specification language while assisting the developers with repetitive tasks. The knowledge base is built up from a NL requirements document in XML in order to capture the contextual information from the document while handling the ambiguity problem and to optimize the process of its translation into a TLG specification with the aid of domain-specific knowledge in XML. Due to its NL-like flexible syntax without losing its formalism, TLG is chosen as a formal specification to fill the gap between the different level of formalisms of NL and formal specification language.

The system is working for some small examples such as the requirements for an Automatic Teller Machine (ATM), with associated banking system domain knowledge. We are performing evaluations of the system for various, more complex, requirements documents, such as the CARA Infusion Pump Controller. The system has been useful in identifying problems and ambiguities with such specifications and in identifying additional information necessary to complete the implementation. It is expected that the technology we are developing will be applicable to these requirements documents as well.

If successful, this will provide a very useful tool to assist software engineers in moving from the requirements document to the formal specification. Our future work is to continue developing the system to improve usability and robustness with respect to its coverage of requirements documents. When finalized, it is expected that by using the formalized context in NLP and TLG as a bridge between the requirements document and a formal specification language, we can achieve an executable and reusable NL specification for a rapid prototyping of requirements, as well as development of a final implementation assisting the developers throughout the software development life cycle.

Acknowledgements. This material is based upon work supported by, or in part by, the U.S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350 and by the U. S. Office of Naval Research under award number N00014-01-1-0746. The authors would like to thank IFAD for providing an academic license to the IFAD VDM Toolbox in order to conduct this research.

References

1. Pohl, K.: The Three Dimensions of Requirements Engineering. Conference on Advanced Information Systems Engineering (1993) 275–292
2. Davis, A.: Software Requirements Analysis and Specification. Prentice-Hall (1990)

3. Boehm, B.W.: Software Engineering Economics. IEEE Transactions on Software Engineering **10** (1984) 4–21
4. Wilson, W.M.: Writing Effective Natural Language Requirements Specifications. Technical report, Naval Research Laboratory (1999)
5. Fuchs, N.E., Schwitter, R.: Attempto Controlled English (ACE). Proc. CLAW 96, 1st Int. Workshop Controlled Language Applications (1996)
6. Wilson, W.M., Rosenberg, L.H., Hyatt, L.E.: Automated Quality Analysis Of Natural Language Requirement Specifications. Technical report, Naval Research Laboratory (1996)
7. Girardi, M.R.: Classification and Retrieval of Software through their Description in Natural Language. PhD thesis, Computer Science Department University of Geneva, Switzerland (1996)
8. Jurafsky, D., Martin, J.: Speech and Language Processing. Prentice-Hall (2000)
9. Bryant, B.R., Lee, B.S.: Two-Level Grammar as an Object-Oriented Requirements Specification Language. Proc. 35th Hawaii Int. Conf. System Sciences (2002) `http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf`
10. Dürr, E., van Katwijk, J.: VDM++ - A Formal Specification Language for Object Oriented Designs. Proc. CompEuro '92 (1992) 214–219
11. Bjørner, D., Jones, C.B.: The Vienna Development Method: The Meta-Language. Springer-Verlag (1978)
12. Quatrani, T.: Visual Modeling with Rational Rose 2000 and UML. Addison-Wesley (2000)
13. IFAD: The VDM++ Toolbox User Manual. Technical report, IFAD (`www.ifad.dk`) (2000)
14. Walter Reed Army Institute for Research (WRAIR): CARA Specification: Roprietary Document. Technical report, WRAIR, Dept. of Resuscitative Medicine (2001)
15. Decker, S., Melnik, S., van Harmelen, F., Fensel, D., Klein, M.C.A., Broekstra, J., Erdmann, M., Horrocks, I.: The semantic web: The roles of XML and RDF. IEEE Internet Computing 4 (2000) 63–74
16. Cleaveland, J.C.: Program Generators with XML and Java. Prentice-Hall (2001)
17. Lakemeyer, G., Nebel, B.: Foundations of knowledge representation and reasoning. Volume 810. Springer-Verlag Inc. (1994)
18. Grady, W.: Moby Part-of-Speech II (data file) (1994)
19. Lee, B.S., Bryant, B.R.: Contextual Knowledge Representation for Requirements Documents in Natural Language. Proc. 15th International FLAIRS Conference (2002) 370–374
20. Bryant, B.R.: Object-Oriented Natural Language Requirements Specification. Proc. ACSC 2000, 23rd Australasian Comp. Sci. Conf. (2000) 24–30

Formal Specification of Non-Functional Aspects in Two-Level Grammar *

Chunmin Yang Beum-Seuk Lee Barrett R. Bryant Carol C. Burt

Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL 35294-1170, U. S. A.
{yangc, leebs, bryant, cburt}@cis.uab.edu

Rajeev R. Raje Andrew M. Olson

Department of Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202, U. S. A.
{rraje, aolson}@cs.iupui.edu

Mikhail Auguston

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, U. S. A.
mikau@cs.nmsu.edu

Abstract

In the UniFrame project, non-functional aspects of distributed software systems are described informally in natural language based on a quality of service (QoS) parameter catalog. Then the descriptions are automatically translated into specifications in a formal specification language, Two-Level Grammar (TLG). The result is a formal QoS specification for rapid prototyping of non-functional aspects of a system as well as their efficient distribution.

Keywords: Formal Specification, Non-functional properties, Quality of Service, Two-Level Grammar (TLG), UniFrame, Vienna Development Method (VDM)

1 Introduction

With the rapid development and increased demand for software systems implemented on computer networks, distributed computing has become the focus of research interest. Even though many techniques have been developed for this purpose most of them focus mainly on the functional aspects of the system neglecting the non-functional aspects. It has been more and more realized that non-functional properties are as important as the functional ones for a successful software product.

The non-functional aspects of software systems are not so much emphasized as the functional aspects due to several reasons:

*This material is based upon work supported by, or in part by, the U.S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350 and by the U. S. Office of Naval Research under award number N00014-01-1-0746.

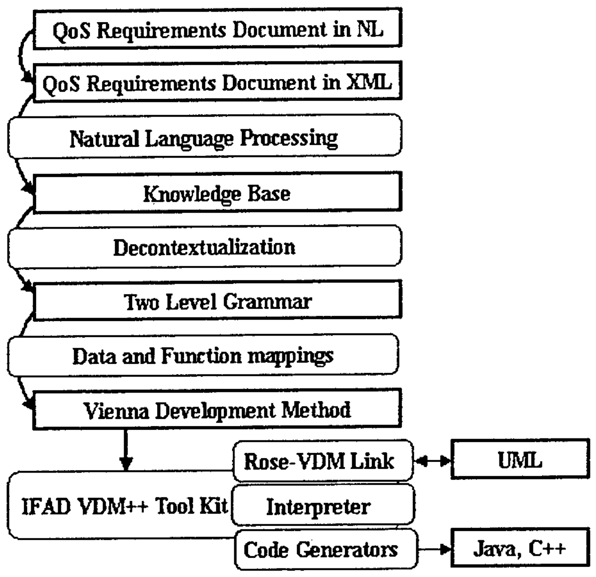

QoS Requirements Document in NL
QoS Requirements Document in XML
Natural Language Processing
Knowledge Base
Decontextualization
Two Level Grammar
Data and Function mappings
Vienna Development Method
IFAD VDM++ Tool Kit
Rose-VDM Link
Interpreter
Code Generators
UML
Java, C++

Figure 1: System Structure.

1. The developers are more concerned with the functionality of the software product than its quality. Their main goal is first to make sure the software is able to provide the functionality as specified by the users. With the move toward component-oriented development, functionality by itself is not enough to meet the users' expectations. To develop a software with high quality, both the functional and the non-functional aspects of the software have to be considered with care.

2. Unlike functional aspects of the specifications, non-functional aspects of the specifications are usually described in an abstract and non-quantified way, thus making it more difficult to describe formally.

3. Non-functional aspects of the specification are complex. Some of the non-functional properties may interact with other non-functional properties. Therefore the effect of non-functional properties of the system does not remain the same all the time, but rather change dynamically according to other non-functional properties.

4. Unlike functional properties, it is difficult to formally specify non-functional properties, although there have been several research projects with this goal, e.g. Aster [1], Qedo [13], QuO [12], to name a few.

Our goal is to enable non-functional requirements to be described informally in natural language and then automatically translated into a formal specification for use in validating component-based software system quality. In our project, first Quality of Service (QoS) requirements in natural language (NL) are represented using eXtensible Markup Language (XML) [3] element and attribute notations which specify the types of non-functional properties and attributes (meta information). This XML specification is translated into a Knowledge Base using Natural Language Processing. Knowledge Base contains the linguistic information as well as meta information of the QoS description. This Knowledge Base is then converted into Two-Level Grammar (TLG) [4] using the collected information in the Knowledge Base. TLG is a formal specification language that is flexible in its natural-language like syntax without losing its formalism. The non-functional specifications in TLG in turn can be translated into VDM++ [7] (an object-oriented extension of the Vienna Development Method [2]) using data and function mappings. VDM++ is chosen as the target specification language because VDM++ has many similarities in structure to TLG and also has a good collection of tools for analysis and code generation. Once the VDM++ representation of the specification is acquired we can do prototyping of the specification using the VDM++ interpreter. Also we can convert this into a high level language such as Java^{TM} or C++ or into a model in the Unified Modeling Language (UML) [14] using the VDM++ Toolkit [8]. The entire system structure is shown in Figure 1. In this paper, we mainly focus on the mechanism to formally specify non-functional aspects of a system in TLG followed by brief illustration of the conversion process from the QoS descriptions in NL into TLG.

2 Quality of Service (QoS)

Quality of Service is a concept originated in the networking area and now it has been extended to software development, in which it is also referred to as "non-functional properties."

To describe the properties of a software product, we need to consider both the functional and non-functional aspects. The former is very straightforward and describes what the software is expected to do. The latter describes how the functions are exhibited. Functional aspects, in practice, earn more attention than the non-functional aspects. From the users' point of view, whether the software can provide the functions as expected is the main issue. More over it is usually easy to prototype or to verify the functionality of the system. On the other hand, it is not so easy to measure the non-functional properties. Functional properties typically have localized effects in the sense that they affect only the part of software functionality whereas non-functional properties specify global constraints that must be satisfied by the software such as performance, fault-tolerance, availability and security.

Along with the development of software engineering techniques, the non-functional properties of a software product become more and more important criteria in classifying a good software product from a poor one since most of the software would successfully provide the required functionality. Therefore the product with non-functional properties will dominate the ones without them. At the same time, there is an increasing demand for fault-tolerance, multimedia, real-time, and other high quality applications, thus the requirement for non-functional properties will become an essential part of software development.

To describe and analyze the non-functional properties, we divide them into three aspects: non-functional attributes, non-functional actions, and non-functional properties. Non-functional attributes are the features or characteristics to be described. A significant characteristic of a non-functional attribute is its decomposability, i.e., a non-functional attribute could be decomposed into multiple more detailed non-functional attributes. Non-functional actions are the input from the outside world which has effect on the attributes. Non-functional properties are the constraints of non-functional actions over the non-functional properties.

This work, to formally specify the quality of components and component complexes (results of compositions of components), is a part of the UniFrame project [16] in which the aspects of a meta-model will be specified and verified in the context of combining heterogeneous components, and provides a QoS management to the interactions between clients and services for distributed object systems by supporting frameworks for multiple QoS categories.

In the project, three steps are taken to assure the QoS of a Distributed Computing System (DCS): first, creation of a catalog for the QoS parameters, then provision of a formal specification of these parameters, and construction of a mechanism for ensuring these parameters, both at the individual component level and at the entire system level.

A catalog of Quality of Service parameters is proposed in [15] which contains the parameters such as throughput, capacity, end-to-end delay, parallelism constraints, availability, ordering constraints, error rate, security, transmission, adaptivity, evolvability, reliability, stability, result, achievability, priority, compatibility, and presentation. The format of this catalog is based on the format of the design patterns catalog. Each parameter is described according to the following features: name, intent, description, influencing factors, measure, known usages, aliases, related parameters, consequences, levels, technologies, applications, exceptions, and example scenario.

There are some reasons that non-functional properties are not explicitly described. First of all, non-functional parameters are more difficult than functional parameters to be dealt with in the sense that they are far more abstract and more complex than the functional parameters. For example, the requirement description may have a phrase like "the system should have very high level of security". But what level of security is considered to be "high?" How can we verify if this system meets this requirement? Obviously, this ambiguous and very inexact description is not descriptive enough to be used as the specification on which the software is developed. In addition, non-functional aspects of the software specification are rarely supported by computer languages, methodologies, or tools [12]. They are usually specified in an informal way and in most cases, they are not quantified thus are more difficult to manipulate. Moreover, it is especially hard to formulate the non-functional aspects of software at early stages of software development. It is not easy to prototype if the system meets the non-functional requirements until the software development phase, thus it is even harder to validate the non-functional properties of a software product. Lastly, the non-functional attributes may conflict or interact with each other. This is called correlation among attributes. When a non-functional action is performed on a system adjusting one non-functional attribute, it may have effects on other non-functional attributes as well. Even though the effect may be unexpected it has to be foreseen and controlled by the software developers.

Although QoS and its guarantees have been widely used in networking, not many attempts have been made to incorporate QoS into component-based software systems [6]. As described above, the informal and ambiguous natural language is not enough for this purpose, and on the other hand, by nature of specification, a programming language is not appropriate either as it has too much detail involved. Formal specification can overcome the problem of natural language being too ambiguous and programming language being too detailed, also formal specification languages have a friendly interface with component based software development techniques, thus our goal is to describe the non-functional properties with such a formal language so as to standardize the software development of systems meeting QoS properties.

3 Specification of QoS in TLG

In UniFrame, Two-Level Grammar (TLG) is used to specify the non-functional properties. TLG is a formal specification language, originally developed as a specification language for programming language syntax and semantics, and later used as an executable specification language and as the basis for conversion from requirements expressed in natural language into formal specifications [4]. It is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The combination of natural language and formalization is unique to TLG and also fits the Unified Meta-component Model (UMM) for component description [16] used in UniFrame well.

The name "two-level" in TLG comes from the fact that TLG consists of two Context Free Languages defining the set of type domains and the set of function definitions operating on those domains, respectively. These grammars may be defined in the context of a class in which case type domains define instance variables of the class and function definitions define methods of the class, and they interact with each other to achieve the power of a Turing Machine.

The syntax of TLG class declarations is:

```
class Identifier-1 [extends Identifier-2, ..., Identifier-n].
  instance variable and function declarations
end class [Identifier-1].
```

From this definition, we can see that TLG supports multiple inheritance. The instance variables (also called as meta-rules) comprising the class definition are declared using domain declarations of the following form:

```
Identifier-1, ..., Identifier-m :: data-object-1; ...; data-object-n.
```

where each `data-object-i` is a combination of domain identifiers, singleton data objects, and lists of data objects, which taken together as a union form the type of `Identifier-1, ..., Identifier-m`.

The function signature (referred to as a hyper-rule as well) is defined as follows.

```
function signature : function-call-1, ..., function-call-n.
```

where $n \geq 1$. Function signatures are a combination of NL words and domain identifiers, corresponding to variables in a logic program. Some of these variables will typically be input variables and some will be output variables, whose values are instantiated at the conclusion of the function call. Therefore, functions usually return values through the output variables rather than directly, in which case the direct return value is considered as a Boolean `true` or `false`. `true` means that control may pass to the next function call, while `false` means the rule has failed and an alternative rule should be tried if possible. Alternative rules have the same format as that given above. If multiple function rules have the same signature, then the multiple left hand sides may be combined with a ; separator, as in:

```
function signature :
  FunctionCall-11, FunctionCall-12, ..., FunctionCall-1j;
  FunctionCall-21, FunctionCall-22, ..., FunctionCall-2k;
  ...
  FunctionCall-n1, FunctionCall-n2, ..., FunctionCall-nm.
```

where there are n alternatives, each having a varying number of function calls. Besides Boolean values, functions may return regular values, usually the result of arithmetic calculations. In this case, only the last function call in a series should return such a value.

TLG is a suitable specification language to represent non-functional properties for the following reasons. First of all, TLG has a class hierarchy which corresponds to the way we describe non-functional properties. This class consists of instance variables and functions, just like the non-functional attributes and non-functional actions encapsulated together. Thus meta-rules of TLG can be used to represent the non-functional attributes while hyper-rules of TLG can be used to represent the non-functional actions.

The classes in TLG may inherit from other classes and this hierarchical structure may be used to represent the decomposability of the non-functional properties as mentioned above so as to take advantage of software reuse, an important idea in component-based software development. Furthermore, TLG is natural language like, and thus it is easier to translate from natural language specification to TLG than to other formal specification languages. TLG is also appropriate for the basis of converting from requirement specifications into other formal specification languages.

Lastly the specification with TLG has a high level of abstraction and its representation is flexible - not all the members (variables or functions) have to be quantifiable. For example, to represent the effect of non-functional actions over the non-functional attributes, especially in the case of correlation, we do not have to quantify all the attributes or properties. In most cases, we only need to know if an action has effect on an attribute or not, and how it affects the attribute if it does have effect. So we only need some variables to describe the relationship above: "no effect," or effects in favor of, or against, respectively. These are just variables, and do not indicate how much the action affects the attribute.

A simple ATM (Automated Teller Machine) example is used to illustrate our approach of using Two-Level Grammar to represent non-functional properties. Here is a brief description of the non-functional requirements of ATM:

```
ATM's security property is as follows. The length of the encryption byte should be bigger than 3 and
the allowed attempts has to be smaller than the maximum allowed attempts. If the encryption byte
length is 6 and the maximum allowed attempts is less than 5 then the system is 80% secure. If the account type is
a savings account or the maximum allowed connections of the bank is less than 50 or the delay level is less
than 50 then the maximum allowed attempts is limited to 4.
If the user timeout is between 10000 and 120000 milliseconds we have a good delay level. If the response
time is longer than 30000 milliseconds, the delay level drops down to 40%.
```

To implement the above requirements specification, four classes are declared : `Property`, `Bank_Capacity`, `ATM_Security`, and `ATM_Delay`. In this simple example, only several non-functional properties are indicated. For each class, there are non-functional attribute definitions, and non-functional action declarations, especially the correlated attributes are defined. In general, not all the non-functional attributes need to be defined exhaustively.

```
class Property.
  Level :: int.
end class.

class Bank_Capacity extends Property.
  Maximum_Connections :: Integer.
end class.

class ATM_Security extends Property.

  Maximum_Allowed_Attempts :: Integer.
  Encryption_Byte_Length :: Integer.
  Allowed_Attempts :: Integer.
  Account_Type :: String.

  check satisfaction :
    Encryption_Byte_Length > 3, Allowed_Attempts < Maximum_Allowed_Attempts.

  update level :
    Encryption_Byte_Length = 6,
    Allowed_Attempts < 5,
    Level := 80.
  update attributes :
    Account_Type = "savings", Maximum_Allowed_Attempts := 4;
    Bank_Capacity Maximum_Connections < 50, Maximum_Allowed_Attempts := 4;
    ATM_Delay Level < 50, Maximum_Allowed_Attempts := 4.

end class.

class ATM_Delay extends Property.

  Response_Time :: Integer.
  User_Timeout :: Integer.

  check satisfaction :
    User_Timeout > 10000, User_Timeout < 120000.

  update level :
    Response_Time > 30000, Level := 40.

end class.
```

Each property is defined as a TLG class whereas the non-functional attributes are defined as TLG instance variables such as Level, Maximum_Connections, Maximum_Allowed_Attempts, Encryption_Byte_Length, Allowed_Attempts, Account_Type, Response_Time, and User_Timeout. In our example, ATM has Security and Delay properties and Bank has Capacity property which is used in update attribute operation of ATM_Security. As the above TLG specification illustrates, all the property classes extend the class Property which has the instance variable Level. This variable is a representative value for the property, with which the decomposability of QoS is implemented. For example ATM_Security property has several attributes such as Encryption_Byte_Length and Allowed_Attempts. The value of Level for ATM_Security represents the overall security level after evaluating all the attributes.

Non-functional actions are represented as methods in the classes. In this example, there is a method that checks the level of property satisfaction (check satisfaction), that updates the overall level of the non-functional properties (update level), or that updates the individual attribute according to dynamic changes of other attributes (update attribute).

Attributes may be updated in a method when some conditions hold. These conditions may include not only the attributes in the same property of the same class, but also the attributes of other property or even in other classes. This is how the correlation of non-functional actions are implemented in TLG. For example, in the ATM_Security class above, if any of the following 3 conditions holds, the maximum number of allowed attempts (Maximum_Allowed_Attempts) is set to be 4: the account type (Account_Type) is a savings account, or the maximum number of connection allowed by the bank at one time (Maximum_Connections) (which is an attribute of Bank_Capacity class) is less than 50 connections, or the Level of ATM_Delay is less than 50.

Usually when a non-functional action is performed on a non-functional attribute, the non-functional attributes may change which, in turn, may trigger other actions to take place. In the ATM example, if some non-functional actions change Account_Type (which is an attribute of ATM_Security), Maximum_Connections (which is an attribute of Bank_Capacity), or the Level of ATM_Delay not only they themselves will be updated, but the value of Maximum_Allowed_Attempts will be updated as well according to the specification in the update attributes method in ATM_Security class.

In summary, as illustrated using a simple ATM example, TLG is proven to be a powerful specification language to formally specify non-functional aspects of a system with a mechanism to abstract the decomposability and to express dynamic correlations among properties and attributes.

4 Conversion from Natural Language Description of QoS into TLG

First the natural language description of QoS of the system is represented in XML to specify which role each sentence plays as a non-functional aspect or attribute. This process is carried out by a natural language parser as a preprocessing of the actual translation into TLG. A sample XML representation of ATM example is shown as follows.

```
<document>
<c title = "ATM">
<c title = "Security">
<p meta = "satisfaction check">
<s>The length of the encryption byte should be bigger than 3 and the allowed attempts has to be smaller
    than the maximum allowed attempts</s>
</p>
<p meta = "level update">
<s>If the encryption byte length is 6 and the allowed attempts is less than 5 then the system is 80%
    secure</s>
</p>
<p meta = "attribute update">
<s>If the account type is a savings account or the maximum allowed connections of the bank is less than 50
    or the delay level is less than 50 then the maximum allowed attempts is limited to 4</s>
</p>
</c>
<c title = "Delay">
<p meta = "satisfaction check">
<s>If the user timeout is between 10000 and 120000 milliseconds we have a good delay level</s>
</p>
<p meta = "level update">
<s>If the response time is longer than 30000 milliseconds the delay level drops down to 40%</s>
</p>
</c>
</c>
</document>
```

Titles such as `Security` and `Delay` indicates the property types whereas the meta information such as `satisfaction check`, `level update`, and `attribute update` indicates the non-functional actions within the property.

Given this XML representation of QoS, each sentence of the specification is tokenized and then by using computational linguistic parsing techniques the system constructs its correct parsing tree. This parsing tree contains the linguistic information about the sentence such as the part of speech (e.g. noun, verb, adjective) and the part of sentence (e.g. subject and object) of each word in the sentence. Obtaining this type of linguistic information is important in the later conversion into TLG because usually the subject of the sentence is identified as the component name. The verb normally indicates what kind of action this component takes to affect a specific QoS. Also anaphoric references (pronouns), elliptical compound phrases, comparative phrases, compound nouns, and relative phrases are handled to allow the input natural language description to be as less controlled as possible. The same technique has been used to automatically translate functional requirements documents into a formal specification language as well [10].

Using this linguistic information and the meta information from XML tags, a Knowledge Base is constructed. The Knowledge Base is an explicit and declarative representation that is used to represent, maintain, and manipulate knowledge about QoS of the system. In addition, the knowledge base has to reflect the structure of TLG into which the Knowledge Base is translated later. The Knowledge Base of the ATM example is shown in Figure 2. In the figure, the blank oval indicates `OR` where as the black ovals indicate `AND` relation. The sentences that are grayed out are the conditional statements compared with normal statements.

This Knowledge Base is converted into TLG by identifying the classes, data types, and operations. Once TLG specifications are obtained, the specifications are translated into VDM++ (we refer the readers to [4] for details). Using the VDM++ tool kit [9] the specifications can be in turn translated into a high level language such as Java or C++ or into a model in UML (Figure 3).

In summary, the QoS description in NL is represented in XML to specify the meta information and the Knowledge Base with a systematic structure can be used to capture this meta information as well as the linguistic information to be used to convert the description into TLG.

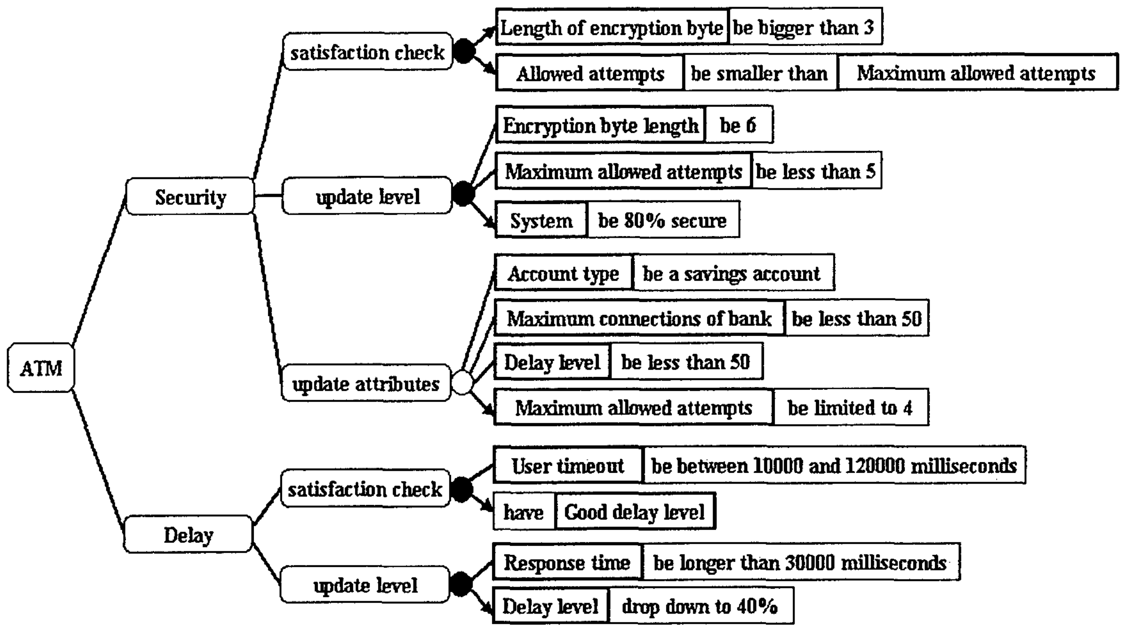

ATM
Security
Delay
satisfaction check
update level
update attributes
satisfaction check
update level
Length of encryption byte | be bigger than 3
Allowed attempts | be smaller than | Maximum allowed attempts
Encryption byte length | be 6
Maximum allowed attempts | be less than 5
System | be 80% secure
Account type | be a savings account
Maximum connections of bank | be less than 50
Delay level | be less than 50
Maximum allowed attempts | be limited to 4
User timeout | be between 10000 and 120000 milliseconds
have | Good delay level
Response time | be longer than 30000 milliseconds
Delay level | drop down to 40%

Figure 2: Knowledge Base for ATM example.

5 Conclusion

Non-functional aspects of the software specification are as important as functional aspects in software development. Formal representation of non-functional aspects is of great contribution to software engineering especially in distributed component-based systems. The specification has to be expressive enough to cover all the useful non-functional specifications while being able to describe complex decomposability and dynamic correlations among the non-functional properties.

In our research first the non-functional specifications are described informally in natural language according to a QoS parameter catalog. Then this specification in natural language is translated into TLG, a natural language like formal specification language. TLG is used to formally represent non-functional aspects of requirements for rapid prototyping and optimal distribution of components. We are performing evaluations of the system for various requirements documents. It is expected that the technology we are developing will be applicable to these requirements documents. If successful, this will provide a very useful tool to assist software engineers in moving from the requirements document to the formal specification.

OMG's Model Driven Architecture (MDA) [11] includes standards that enable the use of generative techniques for construction of interoperability bridges between platform technologies. It will be a promising and useful approach to combine Model Driven Architecture and formal methods in representing the non-functional aspects of software specifications. QoS issues in MDA have been explored in [5]. Our future work is to express the constraints in Object Constraint Language (OCL), and to automatically generate the OCL representation from the TLG representation of the non-functional aspects of software specification, and implement the representation within MDA. At the same time, we will continue developing the system to improve system usability and robustness with respect to its coverage of requirements documents.

References

[1] ASTER. Software Architectures for Distributed Systems (ASTER). Technical report, (`http://www-rocq.inria.fr/solidor/work/aster.html`), 2000.

[2] D. Bjørner and C. B. Jones. *The Vienna Development Method: The Meta-Language.* Springer-Verlag, 1978.

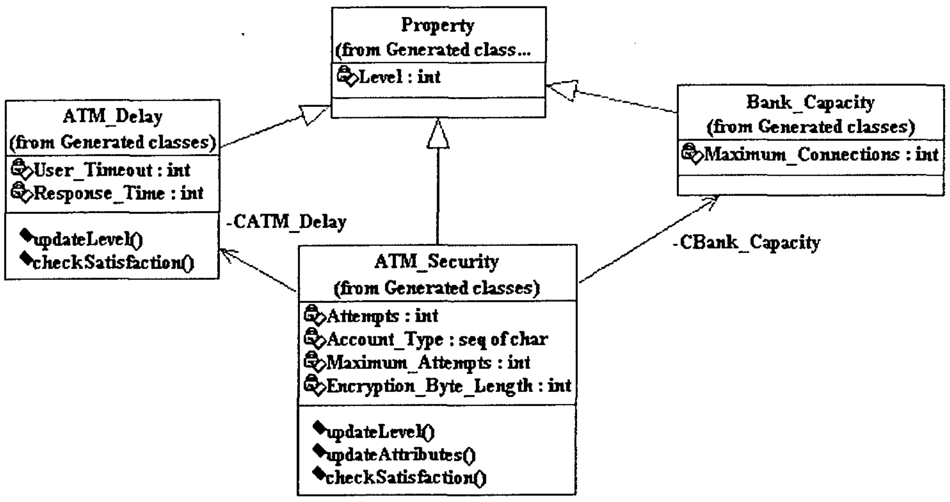


Figure 3: UML for ATM.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, W3C (`http://www.w3c.org/xml`), 2000.

[4] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proc. 35th Hawaii Int. Conf. System Sciences*, Jan. 2002.

[5] C. C. Burt, B. R. Bryant, R. R. Raje, A. Olson, and M. Auguston. Quality of Service Issues Related to Transforming Platform Indepent Models to Platform Specific Models. *Proc. EDOC 2002, 6th IEEE Int. Enterprise Distributed Object Computing Conf. (to appear)*, 2002.

[6] L. A. Campbell and B. H. C. Cheng. Integrating informal and formal approaches to requirements modeling and analysis. *Proc. IEEE International Symposium on Requirements Engineering (RE01)*, pages 294–295, 2001.

[7] E. H. Dürr and J. van Katwijk. VDM++ - A Formal Specification Language for Object-Oriented Designs. *Proc. TOOLS USA '92, 1992 Technology of Object-Oriented Languages and Systems USA Conf.*, pages 63–278, 1992.

[8] IFAD. The VDM++ Toolbox User Manual. Technical report, IFAD (`http://www.ifad.dk`), 2000.

[9] IFAD. VDMTools - Java/C++ Code Generator. Technical report, IFAD, 2000.

[10] B.-S. Lee and B. R. Bryant. Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language. *Proc. ACM 2002 Symposium on Applied Computing*, pages 932–936, 2002.

[11] OMG. Model Driven Architecture (MDA). Technical report, (`http://www.omg.org/mda/`), 2000.

[12] P. Pal, J. Loyall, and R. Schantz et al. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. *Proc. 3rd IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, 2000.

[13] Qedo. QoS Enabled Distributed Objects. Technical report, (`http://qedo.berlios.de`).

[14] T. Quatrani. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 2000.

[15] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, and C. Burt. A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components. *to appear in Concurrency and Computation: Practice and Experience*, 2002.

[16] R. R. Raje, B. R. Bryant, M. Auguston, A. M. Olson, and C. C. Burt. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. *Proc. 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration*, pages 109–119, 2001.

Towards Fully Automatic Execution Monitoring

Clinton Jeffery, Mikhail Auguston, Scott Underwood

Department of Computer Science, New Mexico State University
{jeffery, mikau, sunderwo}@cs.nmsu.edu

Abstract. UFO is a new application framework in which programs written in FORMAN, a declarative assertion language, are compiled into execution monitors that run on a virtual machine with extensive monitoring capabilities provided by the Alamo monitor architecture. FORMAN provides an event trace model in which precedence and inclusion relations define a DAG structure that abstracts execution behavior. Compiling FORMAN assertions into hybrid run-time/post-mortem monitors allows substantial speed and size improvements over post-mortem analyzers. The UFO compiler generates code that computes the minimal projection of the DAG necessary for a given set of assertions. UFO enables fully automatic execution monitoring of real programs. The approach is non-intrusive with respect to program source code and provides a high level of abstraction for monitoring and debugging activities. The ability to compile suites of debugging rules into efficient monitors, and apply them generically to different programs, enables long-overdue breakthroughs in program debugging.

1. Motivation

Debugging is one of the most challenging and least developed areas of software engineering. A special issue of Communications of the ACM characterized the current state of debugging tools as a "Debugging scandal" [1]. According to the classic "Brook's rule" [2] more than 50% of all time and effort in a software project is spent in testing and debugging activities. Typical activities include detection and removal of errors, profiling, and performance tuning.

Debugging activities include queries regarding many aspects of target program behavior: sequences of steps performed, histories of variable values, function call hierarchies, checking of pre- and post-conditions at specific points, and validating other assertions about program execution. Performance testing and debugging involves a variety of profiles and time measurements. Visualization is another common debugging activity that may help locate logic or performance problems.

There is an urgent need for tools that automate the primary, labor-intensive tasks of debugging, but progress has been slow. Debugging automation has its own system of ideas and domain-specific programming activities. Support for these concepts and activities is essential in order to move debugging automation forward.

We are building automatic debugging tools based on precise program execution behavior models that enable us to employ a systematic approach. Our program behavior models are based on events and event traces [3][4][5]. Debugging automation refers to a computation over an event trace. *Program execution monitors* are programs that load and execute a target program, obtain events at run-time, and perform computations over the event trace. Computations are performed during execution, post-mortem, or in any mixture of both times.

Any detectable action performed during a target program's run time is an *event*. For instance, expression evaluations, statement executions, and procedure calls are all examples of events. An event has a beginning, an end, and some duration; it occupies a time interval during program execution. This leads to the introduction of two basic binary relations on events: partial ordering and inclusion. Those relations are determined by target language syntax and semantics, e.g. two statement execution events may be ordered, or an expression evaluation event may occur inside a statement execution event. The set of events produced at run time, together with ordering and inclusion relations, is called an *event trace* and represents a model of program behavior. An event trace forms an acyclic directed graph (DAG) with two types of edges corresponding to the basic relations.

Our previous work included the FORMAN assertion language [3] and the Alamo program execution monitoring architecture [6]. FORMAN takes a top-down approach, introducing a domain-specific syntax for expressing bug manifestations and other behavior of interest, while Alamo takes a more bottom-up, implementation-driven approach, providing runtime system support for the development of monitors in which efficiency and scalability to real programs are primary concerns. Alamo's efficient source-level access and control over monitored programs has been integrated into a production virtual machine; in the absence of such support, monitoring would require extensive low-level instrumentation and control mechanisms.

The language UFO (Unicon-FORMAN) integrates the experience accumulated in these previous projects to provide a complete solution for development of an extensive suite of automatic debugging tools. UFO is an implementation of FORMAN for debugging programs written in the Unicon and Icon programming languages [7][8]. Previous FORMAN implementations worked on subsets of Pascal, and C languages and used post-mortem event trace processing methods that limited their applicability. In contrast, UFO uses the Alamo monitoring architecture that pervades the Unicon virtual machine to support debugging real programs at run time.

2. Unicon and Alamo

The Unicon language and the Alamo monitoring architecture provide the underlying research framework for the implementation of UFO. Unicon is an imperative, goal-directed, object-oriented superset of the Icon programming language. Unicon's syntax is similar to Pascal or Java, while its semantics are higher level, featuring built-in backtracking and heterogeneous data structures and string scanning facilities. Icon has influenced many scripting languages such as Python. Unicon is Icon's direct descendant, derived from Icon's implementation. It runs regular Icon programs and extends Icon's reach with object-orientation and packages, as well as a much richer system interface with high level graphics, networking, and database facilities.

The reference implementation of Unicon is a virtual machine. Virtual machines (VM) are attractive to language implementers, enhancing portability and allowing simpler implementation of very high level language features such as backtracking.

VMs are also ideal for developing debugging tools. VMs provide an appropriate level of abstraction for developing behavior models to describe program executions in a processor independent manner, as illustrated by the JPAX tool [9]. VMs also provide easy access to program state and control flow, the information most needed for debugging activities. Automatic instrumentation on multiple semantic levels is greatly simplified via the use of a VM. This potential was exploited in the Unicon VM by a framework that implements the Alamo monitoring architecture. Event instrumentation and processing support are an integral part of the VM.

The Alamo Unicon framework is summarized in Figure 1. Execution monitors (EM) and the target program (TP) execute as (sets of) coroutines with separate stacks and heaps inside a common VM. The VM is instrumented with approximately 150 kinds of atomic events, each one reporting a <code,value> pair. EMs specify categories of events by supplying an event mask when they activate the TP by coroutine switch. The TP executes up to an event of interest.

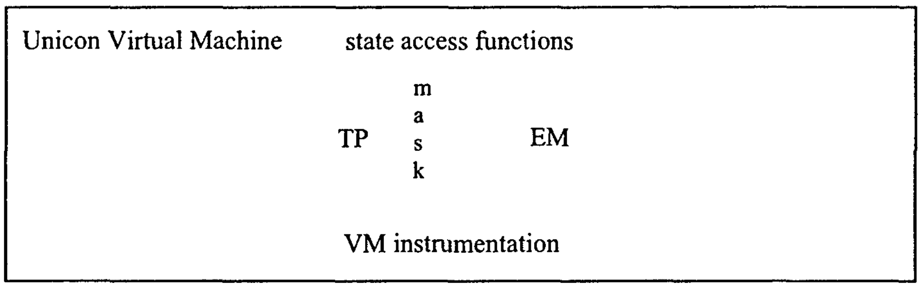


Fig. 1. Alamo architecture within the Unicon VM.

The event mask is used by the VM for instrumentation selection and control. Event reports during TP execution are coroutine context switches from the VM runtime system back to the execution monitor. In addition to the <code,value> reported for the event, the EM can directly access arbitrary variable values and state information from the TP via state access functions. Monitors are written independently from the target program, and can be applied to any target program without recompiling the monitor or target program. Monitors dynamically load target programs, and can easily query the state of arbitrary variables at each event report. Multiple monitors can monitor a program execution, under the direction of a monitor coordinator.

Alamo's goal was to reduce the difficulty of writing execution monitors to be just as easy as writing other types of application programs. UFO moves beyond Alamo to efficiently support FORMAN's more ambitious goal of reducing the difficulty of writing automatic debuggers to the task of specifying generic assertions about program behavior. UFO's FORMAN language is described in Section 4 below, but first it is necessary to present the underlying behavior model.

3. An Event Grammar for Unicon

Event grammars provide a model of program run time behavior. Monitors do not have to parse events using this grammar, since event detection is part of VM and UFO runtime system functionality. Monitors implement computations over event traces supplied by the VM. An event is an abstraction of a detectable action performed at run time and has an event type and various attributes associated with it. The following description in fact provides a "lightweight" semantics of the Unicon programming language tailored for specification of debugging activities. An event corresponds to some specific action of interest performed during program execution. Event type is an important part of the behavior model.

Universal attributes are found in every event. They frequently are used to narrow assertions down to a particular domain (function, variable, value) of interest. Some of these attributes are much easier to obtain than others, and affect the optimizations that can be performed when generating monitor code; see Section 5 for details.

source_text:	in a canonical form
line_num, col_num:	source text locations
time_at_end, time_at_begin, duration:	timing attributes
eval_at_begin (Unicon-expr), eval_at_end (Unicon-expr):	runtime access to the program states
prev_path, following_path:	set of events before/after this event

Event types and their type-specific attributes are summarized in the table below.

Event Type	Description	Type Specific Attributes
prog_ex	whole program execution	
expr_eval	expression evaluation	value, operator, type, failure_p
func_call	function call	func_name, paramlist
input, output	I/O	file
variable	variable reference	
literal	reference to a constant value	
lhp	lefthand part, assignment	address
rhp	righthand part, assignment	
clause	then-, else-, or case branch execution	
test	test evaluation	
iteration	loop iteration	

Event types form a hierarchy, shown in Figure 2. Subtypes inherit attributes from the parent type. Expression evaluation is the central action during Unicon program execution, this explains why the expr_eval event is on the top of the hierarchy.

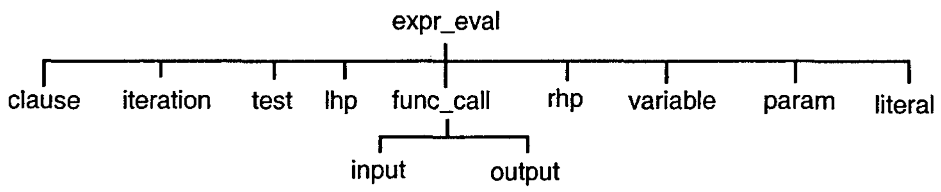


Fig 2. Event Type Inheritance Hierarchy

The UFO *event grammar* for Unicon is a set of axioms describing the structure of event traces with respect to the two basic relations: inclusion and precedence. The grammar is one possible abstraction of Unicon semantics; other event grammars with far more (or less) detail might be used. The event grammar limits what kinds of bugs can be detected, so some detail is useful. The grammar uses the following notation:

Notation	Meaning
A :: (B C)	B precedes A, A includes B and C
A*	Zero or more A's under precedence
A+	One or more A's under precedence
A | B	Either A or B; alternative
A?	A is optional
{ A , B }	Set; A and B have no precedence
x : A	Let x denote event A

```
prog_ex::   ( expr_eval *)
expr_eval:: (  ( expr_eval ) |            unary op
               ( expr_eval  expr_eval ) | binary op
               ( expr_eval+ ) |
               ( test clause ) |          conditional / case expressions
               ( iteration * ) |          loops
               ( { lhp,  rhp} )           assignment
                                          lhp and rhp are not ordered, beginning of
                                          lhp precedes rhp, and end of lhp follows rhp
            )
iteration::  ( test expr_eval*) | ( expr_eval* test ) | ( expr_eval * )
```

Execution of a Unicon program produces a set of events (an *event trace*) organized by precedence and inclusion into a DAG. The structure of the event trace (event types, precedence and inclusion of events) is constrained by the event grammar axioms above. The event trace models Unicon program behavior and provides a basis to define different kinds of debugging activities (assertion checking, debugging queries, profiles, debugging rules, behavior visualization) as appropriate computations over the event traces.

4. FORMAN

Alamo allows efficient monitors to be constructed in Unicon, but using a special-purpose language such as FORMAN, with the rich behavior model described in the preceding section, has compelling advantages. On a basic level, for example, it is convenient to refer to target program variables directly instead of through a library call. For example, in FORMAN we may refer to target program variable x, while in the Unicon monitor it is referenced as variable("x", &eventsource). UFO rules are up to an order of magnitude smaller (in terms of lines of source code) than the equivalent imperative monitors written in Unicon, depending on the type of quantifiers and aggregate operations used in the FORMAN rule.

More important than such conveniences are FORMAN's control structures that directly support dynamic analysis. FORMAN supports computations over event traces centered around event patterns and aggregate operations over events. The simplest event pattern consists of a single event type and matches successfully an event of this type or an event of a subtype of this type. Event patterns may include event attributes and other event patterns to specify the context of an event under consideration. For example, the event pattern

```
E: expr_eval  & E.operator == ":="
```

matches an event of assignment. Temporary variable E provides an access to the events under consideration within the pattern.

The following example demonstrates the use of an aggregate operation.

```
CARD[A: func_call & A.func_name == "read" FROM prog_ex]
```

yields a number of events satisfying the given event pattern, collected from the whole execution history. Expression [...] is a list constructor and CARD is an abbreviation for a reduction of '+' operation over the more general list constructor:

```
+/[A:func_call & A.func_name=="read" FROM prog_ex APPLY 1]
```

Quantifiers are introduced as abbreviations for reductions of Boolean operations OR and AND. For instance,

```
FOREACH Pattern FROM event_set Boolean_expr
```

is an abbreviation for

```
AND/[Pattern FROM event_set APPLY Boolean_expr ]
```

Debugging rules in FORMAN usually have the form:

```
Quantified_expression
     WHEN SUCCEEDS SAY-clauses
     WHEN FAILS SAY-clauses
```

The Quantified_expression is optional and defaults to TRUE. The execution of FORMAN programs relies on the Unicon monitors embedded in a VM environment. Section 5 below describes the architecture of the UFO compiler and runtime system, which translates FORMAN to Unicon VM monitor code.

The following examples illustrate additional features of FORMAN as needed.

Application-Specific Analyses

This section presents formalizations of typical debugging rules. UFO supports and improves upon the most common application-specific debugging techniques. For example, UFO supports traditional precondition checking, or print statement insertion, without any modification of the target program source code. This is especially valuable when the precondition check or print statement is needed in not just one location, but instead in many locations scattered throughout the code.

Example #1: Tracing. Probably the most common debugging method is to insert output statements to generate trace files, log files, and so forth. This allows for subsequent human analysis, and while it has its limitations, it will remain a common technique. It is possible to request evaluation of arbitrary Unicon expressions at the beginning or at the end of events. The VM evaluates these expressions at the indicated time moments, allowing dynamic instrumentation of the Unicon program, whether to print some values, or to call a visualization library subroutine.

```
FOREACH A: func_call & A.func_name == "my_func" FROM prog_ex
  A.value_at_begin(write("entering my_func, value of X is:",X))
AND
  A.value_at_end(write("leaving my_func, value of X is:", X))
```

This debugging rule causes calls to `write()` to be evaluated at selected points at run time, just before and after each occurrence of event A.

Example #2: Profiling. A myriad of tools are based on a premise of accumulating the number of times a behavior occurs, or the amount of time spent in a particular activity or section of code. The following debugging rule illustrates such computations over the event trace.

```
SAY("Total number of read() statements: "
    CARD[ r: input & r.filename == "xx.in" FROM prog_ex ]
    "Elapsed time for read operations is: "
    SUM [ r: input & r.filename == "xx.in" FROM prog_ex
          APPLY r.duration] )
```

Example #3: Pre- and Post- Conditions. Typical use of assertions includes checking pre- and post-conditions of function calls.

```
FOREACH A:func_call & A.func_name=="sqrt" FROM prog_ex
  A.paramlist[1] >=0 AND
  abs(A.value*A.value-A.paramlist[1]) < epsilon
WHEN FAILS SAY("bad sqrt(" A.paramlist[1] ") yields " A.value)
```

Generic Bug Descriptions

Another interesting prospect is the development of a suite of generic automated debugging tools that can be used on any Unicon program. UFO provides a level of abstraction sufficient for specifying typical bugs and debugging rules.

Example #4: Detecting Use of Un-initialized Variables. Although reading an un-initialized variable is permissible in Unicon, this practice often leads to errors. Therefore, in this debugging rule all variables within the target program are checked to ensure that they are initialized before they are used.

```
FOREACH V: variable  FROM prog_ex
  FIND D: lhp FROM V.prev_path   D.source_text == V.source_text
WHEN FAILS SAY( " uninitialized variable " V.source_text)
```

Example #5: Empty Pops. Removing an element from an empty list is representative of many expressions that fail silently in Unicon. While this can be convenient, it can also be a source of difficult to detect logic errors. This assertion assures that items are not removed from empty lists.

```
FOREACH a:func_call & a.func_name=="pop" &
                      a.value_at_begin(*a.paramlist[1]==0)
  SAY("Popping from empty list at event " a)
```

5. Implementation Issues

The most important of implementation issues is the translation model by which FORMAN rules are compiled into Unicon monitors. Rules are written as if they have the complete post-mortem event trace available for processing. This generality is powerful; however the majority of assertions can be compiled into monitors that execute entirely at runtime. Runtime monitoring is the key to practical implementation. For assertions that require post-mortem analysis, the UFO runtime system computes a projection of the execution DAG needed to perform the analysis.

The UFO translation model categorizes each rule as either "runtime", "post-mortem", or "hybrid", denoting the amount of computations that can be performed at runtime. Runtime and hybrid categories are determined by constraints on FORMAN quantifier prefixes and result in more efficient code. Nested quantifiers and aggregate operations generally require post-mortem operation.

Translation Examples

Each FORMAN statement is translated into a combination of initialization, run-time, and post-mortem code. Monitors are executed as coroutines with the Unicon target program, as explained in Section 2. The following examples give a flavor of the run time architecture of monitors generated from the UFO high level rules.

Implementation of Example #1. A lone FOREACH quantifier is typical of many UFO debugging actions and allows computation to be performed entirely at runtime. The events being counted and values being accumulated determine an *event mask* in the initialization code that defines the Alamo events that will be monitored. The monitor's event processing loop implements a filter based on procedure name within an if-expression. The Unicon code blocks containing write() expressions are inserted directly into the event loop for the relevant events. The complete monitor is:

```
$include "evdefs.icn"
link evinit
procedure main(av)
  EvInit(av) | stop("can't monitor ", av[1])
  mask := E_Pcall ++ E_Pret ++ E_Pfail ++ E_Prem
  while EvGet(mask) do {
    if &eventcode == E_Pcall & &eventvalue === my_func then
      write("entering my_func, value of X is:", X)       # BEFORE
    if &eventcode == (E_Pret | E_Pfail | E_Prem) &
       &eventvalue=== my_func then
      write("leaving my_func, value of X is:", X)        # AFTER
    }
end
```

Implementation of Example #2. Another typical situation involves an aggregate operation and selection of events according to a given pattern. The SAY expression is implemented by a call to write(); it must be performed post-mortem since it uses parameters whose values are constructed during the entire program execution. CARD denotes a counter, while SUM denotes an accumulator +/; both require a variable that is initialized to zero. The event subtypes and constraints are used to generate additional conditional code in the body of the event processing loop. Lastly, some attributes such as r.duration require additional events and measurements besides the initial triggering event. In the case of r.duration, a time measurement between the function call and its return is needed.

```
$include "evdefs.icn"
link evinit
procedure main(av)
  EvInit(av) | stop("can't monitor ", av[1])
  cardreads := sumreadtime := 0
  mask := cset(E_Fcall)
  while EvGet(mask) do {
    ### count CARD of r:input...
    if &eventcode == E_Fcall & &eventvalue === (read|reads) then
      cardreads +:= 1
    ### add SUM of r.duration for r:input
    if &eventcode == E_Fcall & &eventvalue===(read|reads) then {
      thiscall := &time
      EvGet(E_Ffail++E_Fret)
      sumreadtime +:= &time - thiscall
      }
    }
  ### Translation of SAY
  write("Total number of read() statements: ", cardreads, "\n",
        "Elapsed time for read operations is: ",  sumreadtime)
end
```

Basic Generation Templates

The preceding handwritten example monitors use a single main loop that implements traditional event-driven processing. Monitors generated by the UFO compiler reduce complex assertions to this same single event loop. Keeping event detection in a single loop allows uniform processing of multiple event types used by multiple monitors. The code generated by the UFO compiler integrates event detection, attribute collection, and aggregate operation accumulation in the main event loop.

Assertions in UFO that use nested quantifiers entail two nested loops. Code generation flattens this loop structure, and postpones assertion processing until required information is available. A hybrid code generation strategy performs runtime processing whenever possible, delaying analyses until post-mortem time when necessary. Different assertions require different degrees of trace projection storage; code responsible for trace projection collection is also arranged within the main loop.

Each UFO rule falls in one of the following categories which determines its code generation template in the current implementation. We have not found a use for assertions requiring more than two nested quantifiers.

Type	FORMAN template	Pseudocode
I	Single quantifier. Rule applies to whole trace(prog_ex); evaluates at runtime.	See examples in Section 4.1.

Type	FORMAN template	Pseudocode

II	Nested quantifiers of the form
	Quantifier A: Pattern_A
	    Quantifier B: Pattern_B FROM A
	        Body
	This requires accumulation of a trace projection for B-events and causes a mild overhead at runtime.
		Main Loop
		    Maintain stack of nested A events
		    Accumulate events B in a B-list
		    If end of event A
		        Loop over B-list
		            Do Body
		    Endif
		    If stack of A is empty
		        Destroy B-list
		End of Main Loop

III	Nested quantifiers of the form
	
	Quantifier A: Pattern_A
	    Quantifier B: Pattern_B
	            FROM A .prev_path
	        Body
	Accumulates a trace projection for B-events and may cause a heavy overhead at runtime. The B-list can not be deleted till the end of session.
		Main Loop
		    Maintain stack of nested A events
		    Accumulate events B in a B-list
		    If end of event A
		        Loop over B-list
		            If B precedes A
		                Do Body
		            Endif
		End of Main Loop

IV	Nested quantifiers of the form
	
	Quantifier A: Pattern_A
	    Quantifier B: Pattern_B
	            FROM A .following_path
	        (or FROM prog_ex)
	        Body
	Accumulates trace projections for both A and B events and causes a very heavy overhead at runtime.
		Main Loop
		    Accumulate events A in A-list
		    Accumulate events B in B-list
		End of Main Loop
		# Postmortem Loop
		Loop over A-list
		    Loop over B-list
		        Do Body
		End of Postmortem Loop

Compiler-Based Optimizations

The advantage of the UFO approach is the combination of an optimizing compiler for monitoring code with efficient run-time event detection and reporting. Since we know at compile time all necessary event types and attributes required for a given UFO program, the generated monitor is very selective about the behavior that it observes.

For certain UFO constructs, such as nested quantifiers, monitors accumulate a sizable projection of the complete event trace and postpone corresponding computations until required information is available. The use of the previous_path and following_path attributes in UFO assertions facilitates this kind of optimization.

For further optimization, especially in the case of programs containing a significant number of modules, the following FORMAN construct limits event processing to events generated within the bodies of functions F1, F2, … , Fn.

```
WITHIN F1, F2, … , Fn DO
    Rules
END_WITHIN
```

This provides for monitoring only selected segments of the event trace.

Unicon expressions included in the value_at_begin and value_at_end attributes are evaluated at run time. Some other optimizations implemented in this version are:

- only attributes used in the UFO rule are collected in the generated monitor;
- an efficient mechanism for event trace projection management, which disposes from the stored trace projection those events that will not be used after a certain time (for example, see Category II);
- event types *and* context conditions are used to filter events for the processing.

UFO's goal of practical application to real-sized programs has motivated several improvements to the already-carefully-tuned Alamo instrumentation of the Unicon VM. We are working on additional optimizations.

6. Results of Sample Assertion Execution

Table 1 gives results from executing rules written in UFO on a sample target program, a 1,100 line version of egrep. Tests were run on a 700 MHz Solaris machine with 512MB of RAM. The results reported are number of events generated by the VM and execution time averaged over several runs. Execution time is reported as minutes:seconds.tenths. The second row contains the time for program execution without monitoring. Each program/input file combination was monitored by 8 different assertions corresponding to the basic generation templates.

Cases 1-4 are examples of a Category I template. Case 5 is a Category II rule. Case 6 is a Category III rule. It uses PREV_PATH and accumulates a trace projection over part of the program execution. Cases 7 and 8 contain nested quantifiers that belong to Category IV. These assertions require the accumulation of two trace projections over the entire program execution, and complete post-mortem processing. Case 9 is composed of all the previous assertions to yield a monitor that combines multiple assertions on a single execution of the target program.

Table 1. Results for igrep.icn.

Input Size (lines)	4000		16000		64000	
No monitoring	0.5		1.6		6.4	
	Events	**Time**	**Events**	**Time**	**Events**	**Time**
Case 1	184208	4.1	736208	16.2	2944208	1:04.9
Case 2	284123	4.6	1136123	18.1	4544123	1:12.9
Case 3	184208	3.4	736208	13.5	2944208	54.0
Case 4	184208	3.5	736208	13.6	2944208	54.0
Case 5	276306	6.3	1104306	28.0	4416306	2:09.3
Case 6	276306	6.5	1104306	28.4	4416306	2:11.8
Case 7	276306	6.5	1104306	29.1	4416306	2:11.3
Case 8	276306	6.5	1104306	29.4	4416306	2:12.6
Case 9	340306	45.9	1360306	3:57.8	5440306	20:38.6

The results depicted in this table allow several observations. Average monitoring speeds on simple assertions in the test environment were in the range of 2-3 million events per minute. Monitoring realistic assertions on real-size programs with real-size input data is feasible with this system. Most assertions impose about one order of magnitude execution slowdown compared with the unmonitored program execution.

The execution time required by the combination of all assertions (Case 9) is longer than the sums of separate monitor executions. Combined assertion executions have greater memory requirements in the current implementation, because separately collected trace projections compete for available cache and virtual memory resources. Multi-assertion optimizations are not yet implemented in the current UFO compiler.

7. Related Work

What follows is a very brief survey of basic ideas known in Debugging Automation to provide the background for the approach advocated in this paper.

The Event Based Behavioral Abstraction (EBBA) [10] characterizes the behavior of programs in terms of primitive and composite events. Context conditions involving event attributes are used to distinguish events. EBBA defines two higher-level means for modeling system behavior -- clustering and filtering. Clustering is used to express behavior as composite events, i.e. aggregates of previously defined events. Filtering serves to eliminate from consideration events, which are not relevant to the model being investigated. Both event recognition and filtering can be performed at run-time.

Event-based debuggers for the C programming language built on top of GDB include Dalek [11] and COCA [12]. Dalek supports user-defined events that typically are points within a program execution trace. A target program has to be manually instrumented in order to collect values of event attributes. Composite events can be recognized at run-time as collections of primitive events. COCA uses GDB for tracing and PROLOG for the execution of debugging queries. It provides an event grammar for C and event patterns based on attributes for event search. The query language is designed around special primitives built into the PROLOG query evaluator.

Assertion languages provide another approach to debugging automation. Boolean expressions are attached to points in the target program, like the assert() macro in C. [13] advocates a practical approach to programming with assertions for the C language, and demonstrates that even local assertions associated with particular points within the program may be extremely useful for program debugging

The ANNA [14] annotation language for the Ada language supports assertions on variable and type declarations. The TSL [15], [16] annotation language for Ada uses events to describe the behavior of Tasks. Patterns can be written which involve parameter values of Task entry calls. Assertions are written in Ada using a number of special pre-defined predicates. Assertion-checking is performed at run-time. RAPIDE [17] provides an event-based assertion language for software architecture description. Temporal Rover is a commercial tool for dynamic analysis based on temporal logics [18]. The DUEL [19] debugging language introduces expressions for C aggregate data exploration, for both assertions and queries.

Algorithmic debugging was introduced in [20] for the Prolog language. In [21] and [22] this paradigm is applied to a subset of PASCAL. The debugger executes the program and builds a trace execution tree at the procedure level while saving some useful trace information such as procedure names and input/output parameter values. The debugger traverses the execution tree, asking the user about the intended behavior of each procedure. The search finally ends and a bug is localized within a procedure p when one of the following holds: procedure p contains no procedure calls, or all procedure calls performed from the body of procedure p fulfill the user's expectations. The notion of computation over execution trace introduced in FORMAN is a generalization of Algorithmic Debugging and is a convenient basis for describing such generic debugging strategies.

PMMS [23] is a high level program monitoring and measuring system. This system works by receiving queries from the user about target programs written in the AP5 high level programming language. PMMS instruments the source code of the target program in order to gather data necessary to answer the posed questions. This data is collected during run time by the monitoring facilities of PMMS and stored in a database for subsequent analysis. Their domain specific query language is similar to FORMAN but tailored for database-style query processing.

JPAX [9], the Java Path Explorer, provides a means to check execution events within a program based on a user provided specification written in Maude, a high level logic language. Like UFO, JPAX supports monitoring based on a VM (JVM). JPAX supports both black box (based on automatic byte-code instrumentation) and white box (based on hand instrumentation) runtime verification.

Dynascope [24] is a system for directing programs written in vanilla C. A director monitors and controls the actions of the program, while an interpreter controls the flow of event streams to and from the director in addition to interpreting the program. Dynascope can test and debug programs without altering their source code.

YODA [25] uses a preprocessor to attach statements to a target Ada program. These statements activate a monitor creates a trace database and a symbol table to aid in debugging. The trace database will contain the program's history regarding variable declaration and use, task synchronization, and change in task status. Prolog queries can be issued by the user in order to confirm or reject hypotheses about program behavior. YODA represents a classical post-mortem trace processing paradigm.

8. Conclusions and Future Work

The rising popularity of virtual machine architectures enables dramatic improvements in automatic debugging. These improvements will only occur if debugging is one of the objectives of the VM design, e.g. as in the case of .net [26].

The architecture employed in UFO could be adapted for a broad class of languages such as those supported by the Java VM or the .net VM. Our approach to debugging automation uniformly represents many types of debugging-related activities as computations over traces, including assertion checking, profiling and performance measurements, and the detection of typical errors. We have integrated event trace computations into a monitoring architecture based on a VM. Preliminary experiments demonstrate that this architecture is scalable to real-world programs.

One of our next steps is to build a repository of formalized knowledge about typical bugs in the form of UFO rules, and gather experience by applying this collection of assertions to additional real-world applications. There remain many optimizations that will improve the monitor code generated by the UFO compiler, for example, merging common code used by multiple assertions in a single monitor, and generating specialized VMs adjusted to the generated monitor.

Acknowledgements
This work has been supported in part by U.S. Office of Naval Research Grant # N00014-01-1-0746, by U.S. Army Research Office Grant # 40473--MA-SP, by the NSF Grant # EIA 02-20590, and by the National Library of Medicine.

References

[1] Communications of the ACM, Vol.4, 1997.
[2] F. Brooks, The Mythical Man-Month. Addison-Wesley, Reading, MA, 1975.
[3] Mikhail Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, Proceedings of the 2nd Int'l Workshop on Automated and Algorithmic Debugging, Saint-Malo, France, May 1995, pp. 277-291.
[4] M. Auguston, A. Gates, M. Lujan, "Defining a Program Behavior Model for Dynamic Analyzers", in Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97, Madrid, Spain, June 1997, pp. 257-262.
[5] M. Auguston, "Lightweight semantics models for program testing and debugging automation", Proceedings of the 7th Monterey Workshop, June 2000, pp.23-31.
[6] Clinton L. Jeffery, Program Monitoring and Visualization: an Exploratory Approach. Springer, New York, 1999.
[7] Clinton Jeffery, Shamim Mohamed, Ray Pereda, and Robert Parlett, "Programming with Unicon", http://unicon.sourceforge.net.
[8] Ralph E. Griswold and Madge T. Griswold, The Icon Programming Language, 3rd edition. Peer to Peer Communications, San Jose, 1997.
[9] K. Havelund, S. Johnson, G. Rosu. "Specification and Error Pattern Based Program Monitoring", ESA Workshop on On-Board Autonomy, Noordwijk, Holland, Oct. 2001.
[10] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", Journal of Systems and Software 3, 1983, pp. 255-264.
[11] R. Olsson, R. Crawford, W. Wilson, "A Dataflow Approach to Event-based Debugging", Software -- Practice and Experience, Vol.21(2), February 1991, pp. 19-31.
[12] M. Ducasse, "COCA: An automated debugger for C", in Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, 1999, pp. 504-513.
[13] D. Rosenblum, "A Practical Approach to Programming with Assertions", IEEE Transactions on Software Engineering, Vol. 21, No 1, January 1995, pp. 19-31.
[14] D. C. Luckham, S. Sankar, S. Takahashi, "Two-Dimensional Pinpointing: Debugging with Formal Specifications", IEEE Software, Vol. 8, No 1, January 1991, pp.74-84.
[15] D. C. Luckham, D. Bryan, W. Mann, S. Meldal, D. P. Helmbold, "An Introduction to Task Sequencing Language, TSL version 1.5", Stanford University, Feb. 1990, pp. 1-68.
[16] D. Rosenblum, "Specifying Concurrent Systems with TSL", IEEE Software, Vol. 8, No 3, May 1991, 52-61.
[17] D. Luckham, J. Vera, "An Event-Based Architecture Definition Language", IEEE Transactions on Software Engineering, Vol.21, No. 9, 1995, pp. 717-734.

[18] D. Drusinsky, The Temporal Rover and the ATG Rover, LNCS #1885, pp.323-330, Springer, 2000.

[19] M. Golan, D. Hanson, "DUEL - A Very High-Level Debugging Language", in Proceedings of the Winter USENIX Technical Conference, San Diego, Jan. 1993.

[20] E. Shapiro, "Algorithmic Program Debugging", MIT Press, May 1982.

[21] P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, "Generalized Algorithmic Debugging and Testing", ACM LOPLAS, Vol 1 (4), Dec 1992.

[22] N. Shahmehri, "Generalized Algorithmic Debugging", Ph.D. Thesis No. 260, Dept. of Computer and Information Science, Linköping University, S-581 83 Linköping, 1991.

[23] Y. Liao, D. Cohen, "A Specificational Approach to High Level Program Monitoring and Measuring", IEEE Transactions on Software Engineering, Vol 18, No 11, Nov 1992, pp.969 – 978.

[24] R. Sosic, "Dynascope: a Tool for Program Directing", Sigplan Notices 27(7), pp.12-21, 1992.

[25] LeDoux, Carol H. and Parker, D., "Saving Traces for Ada Debugging. Ada in Use", Proc. of the Ada International Conference, ACM Ada Letters, 5(2), pp.97-108, Sep 1985.

[26] http://www.microsoft.com/net/

Appendix. Syntax for UFO rules

```
Rules::=   ( ( Rule | Within_group ) ';') +
Within_group::= 'WITHIN' Procedure_name ( ',' Procedure_name ) *
      'DO' ( Rule ';' ) + 'END_WITHIN'
Rule::= [ Label ':' ]
     [ ('FOREACH' | 'FIND') Pattern [ 'FROM' 'PROG_EX' ] ]
      [ ('FOREACH' | 'FIND') Pattern [ 'FROM'  ('PROG_EX'  |
                   Metavariable [ '.' ( 'PREV_PATH' | 'FOLLOWING_PATH' )] ) ]
        [ 'SUCH' 'THAT' ]  Bool_expr
        [['WHEN' 'SUCCEEDS'] Say_clause + ]  ['WHEN' 'FAILS'   Say_clause + ]
Say_clause ::= 'SAY' '(' ( Expression | Metavariable | Aggregate_op ) * ')'
Bool_expr::= Bool_expr1 ( 'OR' Bool_expr1 )*
Bool_expr1::= Bool_expr2 ( 'AND' Bool_expr2 )*
Bool_expr2::=  Expr  [ ( '=' | '==' | '>' | '<' | '>=' | '<=' | '|=' ) Expr ] | 'NOT' Bool_expr2 |
                 '(' Bool_expr ')'
Pattern::=   Metavariable ':' Event_type [ '&' Bool_expr ]
Aggregate_op::=  [ ( 'CARD' | 'SUM' ) ] '['  Pattern
  ['FROM' ('PROG_EX' | Metavariable [ '.' ( 'PREV_PATH'|'FOLLOWING_PATH' )] )]
  [ 'APPLY' ( Bool_expr | Expression ) ] ']'
Expression::=   Expr1 (* ( '+' | '-') Expr1 *)
Expr1::= Simple_expr ( ( '*' | 'DIV' | 'MOD' ) Simple_expr )*
Simple_expr::= '-' Simple_expr | integer | Aggregate_op |
                Metavariable '.' Attribute | string | '('  Expr ')'
Attribute::= (SOURCE_TEXT | LINE_NUM | COL_NUM | TIME_AT_END |
          TIME_AT_BEGIN | COUNTER_AT_END | COUNTER_AT_BEGIN |
          DURATION | VALUE | OPERATOR | TYPE | FAILURE | FUNC_NAME |
          ( PARAM_NAMES '[' integer ']' ) | FILE_NAME | ADDRESS |
         ( VALUE_AT_BEGIN | VALUE_AT_END) '(' Unicon_expr ')' )
Event_type::= (func_call | expr_eval | input | output | variable | literal |
          lhp | rhp | clause | iteration | test )
```

A Component Assembly Architecture with Two-Level Grammar Infrastructure[1]

Wei Zhao[2] Barrett R. Bryant[2] Fei Cao[2] Rajeev R. Raje[3] Mikhail Auguston[4]
Andrew M. Olson[3] Carol C. Burt[2]

1. Introduction

Being able to generate a concrete software product from domain specifications, upon an order requirement[5], still remains a mirage using most modern software engineering techniques. To provide a systematic way to automate software engineering process, formal models should be constructed beforehand to capture the various aspects of engineering knowledge for any predictable software solutions for a particular domain; an infrastructure should be available to support the automation of any specific product generation by intelligently using the established engineering knowledge models.

Engineering knowledge involves the decisions made about a software product along its production line, which includes the policies from domain business executives, expertise from domain experts, experiences from software managers and engineers, and the techniques from software developers and programmers. During the software production process, these engineering knowledge will contribute respectively towards service specifications of the system and the Quality of Services (QoS), detailed business logic of the system, specifications of software architecture and role assignments for developers, concrete software development by applying different programming languages and component-based technologies.

Using current software engineering practices, the investments of engineering knowledge are all encapsulated in one business organization, making engineering knowledge implicit, vague and intertwined. However, the latter two aspects are from technology prospective other than business prospective, and can be most possibly reused across all the business domains. To construct formal models that capture various aspects of engineering knowledge, and to organize them in such a way that separation of concern and maximized reuse of engineering knowledge can be achieved, we categorize this synergy of engineering knowledge into three-dimensional domains:

1) Business domains are associated with the natural categorization of business sectors and the natural hierarchical structure of business organizations;

2) Functionality domains are based on the functionality and the role of different parts of software, and their collaboration means and patterns; and

3) Technology domains address the issues related to software implementation technologies such as component models, programming languages, hardware platforms, and so on.

Different group of people or organizations are expected to be responsible for each domain. The successful construction of the Generative Domain Model (GDM) [Cza00] (for each domain

[1] This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant numbers DAAD19-00-1-0350 and 40473-MA, and by the U. S. Office of Naval Research under award number N00014-01-1-0746.

[2] Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, U. S. A., {zhaow, bryant, caof, cburt}@cis.uab.edu.

[3] Department of Computer and Information Science, Indiana University Purdue University Indianapolis, Indianapolis, IN 46202, U. S. A., {rraje, aolson}@cs.iupui.edu.

[4] Computer Science Department, New Mexico State University, Las Cruces, NM 88003, U. S. A., mikau@cs.nmsu.edu.

[5] By "order requirement", we mean either a requirement document written in natural language by human or a system construction request from another computer program.

mentioned above), would assist in automating the development of software products under the guidance of model transformations and refinements from the highest model (GDM) to more specific intermediate models. This would finally lead to the end software products. This paper describes the UniFrame project that aims at this goal.

2. Related Work

Toward the goal of automatic production of software, there have been many attempts in domain engineering, system generation and model transformations. We describe a few prominent ones here.

Generative Programming [Cza00] is well known for providing a vision of automatically generating products from a GDM, a specification of the product domain. However, the examples provided for elementary components envisioned by the authors are limited to C++ **structs** and **classes** with templates, which may not be sufficient to solve problems on the scale of distributed and component-oriented computing. Many problems like universal interoperability should be solved during system integration and generation. Widely known efforts such as CORBA [Corba], Web Services [W3C] and Model Driven Architecture (MDA) [OMG01], an initiative of the Object Management Group (OMG), arose as possible solutions for the interoperability problem.

MDA sketches out a model transformation series, which transforms a business model to a Platform Independent Model (PIM), then to a Platform Specific Model (PSM), and finally gets to the executable code. Steps of model transformations certainly contribute to the automated product generation from the high level specifications. Nevertheless, MDA currently appears to be only concentrating on the model transformation for a single system. It also does not specify the assembly of a system out of many available components.

FORM [Kan98] provides methods to construct feature models for a domain during the domain engineering phase and to generate concrete systems by applying feature selection during the application engineering phase. FORM defines domain features in terms of services, domain technologies, operating environments and implementation techniques. We do consider it to be inappropriate that the feature models for a business domain should include the latter two, as it is not a good practice of separation of concerns, and can be a further obstacle for system flexibility evolvability and engineering knowledge reuse. The architecture defined in FORM from three different viewpoints (subsystem, process and module) does not capture the multi-dimensions of engineering knowledge during a product manufacturing process.

3. UniFrame Architecture Overview

The UniFrame project[6] is a framework for:

1) Providing an architecture for automated software product generation, upon an order requirement, based on the assembly of a selection from an ensemble of searched components (with which we believe we can overcome limitations mentioned in section 2);
2) Providing a practical technique based on the formalism of Two-Level Grammar (TLG) [Bry02], which serves as the infrastructure enabling the automation of software production by steps of model transformations.

UniFrame has two levels:

[6] This project goal statement is phased according to our newly developed ideas and is different from the original official one, which is "Seamless integration of heterogeneous and distributed software components" [UniFr].

- GDMs for business domain, functionality domain and technology domain jointly comprise the core part of the system level of UniFrame. The GDM for the business domain mainly contains: domain feature models, standardized elementary domain service[7] specifications uniquely identified by their Universal Resource Identifiers (URIs), associated Quality of Service (QoS) parameters, service collaboration patterns, typical computing algorithms for this business domain, domain specific language, etc. Some preamble of a business GDM may be a standardized **Stack** class provided by J2SE [J2SE] for the domain of the object "stack", or OpenGL [OpenG] for the domain of graphics and images processing. The functionality domain GDM is essentially a reference architecture model that identifies the functionality, the role and the collaboration patterns among different parts of software. The GDM for technology domain deals with the interoperability across heterogeneous implementation technologies and programming languages. The UniFrame system level sets the context for developing a family of products. We propose an Internet Component Exchange and Assembly (ICEA)[8] center for each business domain for developing and maintaining the business GDM.
- The UniFrame component level gives the view of the single system development. Component developers have the freedom of choosing any implementation technology, underlying hardware platform, or programming language to implement any standardized service or a group of services confining to the service specifications in the business GDM. The developer even has the freedom to name services as long as a DNS server (specialized in this context) can perform the correct translation to the one with standardized semantics and unique URI in the business GDM. Upon the accomplishment of the individual component development, developers need to fill out a Unified Meta-Component Model (UMM) [Raj01] form to formally describe the components. UMM identifies the niches of this component in various GDMs, provides the QoS of this component and the address of the native component registry (e.g. RMI registry if this component is developed in RMI). Then the developers need to register the UMM to its respective ICEA. Hopefully, in the future, this process can be further facilitated by MDA techniques: the developers pick up the business model for any business services, and apply the model transformations to get to the executable code.

TLG is used to represent the three GDMs and the UMM. Because the domain services are standardized and factored, it is feasible for the users to explicitly identify the service semantics in their order requirements. The automatic production is carried out by the joint-effort among the feature models in the three GDMs, feature selection from order requirements, and feature identification and concretization in the UMM. At the system generation time, we need apply the service interaction patterns from the feature models in business GDM for homogeneous components; if the components are heterogeneous, we need apply the component interaction patterns from the functionality GDM, and then use the mapping and translation rules stored in the technology GDM for building interoperability. More precisely for interoperation, the UMM specification (in TLG) will be translated into WSDL [W3C][Cao02], making Web Services the underlying communication technique. The model transformation computation supporting product automation is performed by the TLG interpreter that computes steps of substitution (the first level context-free grammar) between two models (grammar's left and right hand side) guided by the transformation rules (the second-level context-free grammar). Different levels of models will be

[7] The "service" is not an executable entity. It is a concept of a slot of domain businesses. The "component" defined under a component model can be executed within a component framework. The component developers build software components by concretizing services. The "component" is a technologic carrier for "services".
[8] It is our notion of a group of people or organizations for this purpose.

represented by groups of TLG classes, e.g. Class Withdraw is a service description in the bank domain GDM.

```
class Withdraw.
   Passin :: AccountNumber, Amount.
  .....
end class.
```

A lower level model could further define AccountNumber as:

```
class AccountNumber.
     Type :: Integer.
      Language :: c++.
      .....
end class.
```

Or as:

```
class AccountNumber.
     Type :: String.
     Language :: java.
     Lexeme :: letter (letter | number )*.
      .....
  end class.
```

Please refer to [Bry02] for more details on TLG, and refer to [Zha02] for our current definition and examples of TLG as an executable code generator.

4. Engineering Principles Employed in Designing UniFrame

Various engineering principles are observed in designing UniFrame architecture to fulfill its goal:

- Modularity is the fundamental consideration in designing UniFrame. In UniFrame, the final system (product) is built from components, which in turn are built around one or more services. The atomic and factored services (or features) is the truth that the system can be **generated** on demand from requirements, in another word, across all the products of a product family, what can really be reused and re-structured are the elementary services. Given all the possible elementary services for a business domain, a wide spectrum of systems can be generated by various combinations of services. Service composition rules (e.g. domain feature models) are embedded in the business GDM, and the component composition rules [Sun02] are embedded in the functionality GDM.
- The principle of autonomy and separation of concerns naturally separates the multidimensional engineering knowledge into three GDMs maintained by different groups of people, respectively. On the maturity of UniFrame, we hope the stabilized infrastructure will have three sets of APIs that will enable the creation/maintenance of these three GDMs. The experts in different domains have the freedom of controlling their domains; the component developers have their own choice about the implementation details. This makes UniFrame flexible, dynamically re-configurable and evolvable.
- UniFrame also supports a transparent communication channel. The business GDM with standardized services and their QoS is the communication media among the users, the system and the component developers, which ensures what the component developer supplies and what the system produces is exactly what the users want. It also suggests that the automated production could start from as early as order requirements.

- Reflection and intelligent reasoning of model transformations with minimum human interaction is also a key attribute of UniFrame. UMM, a reflection of a component, together with three GDMs provide the TLG-facilitated infrastructure enough knowledge to pursue intelligent reasoning in the process of system assembly, e.g. automatically reason about component properties and relationships.

5. Two-Level Grammar

As UniFrame maturates, the infrastructure is not intended for frequent human manipulations. It is reasonable to choose TLG (textual with functional and logic programming language style) as the machine-understandable infrastructure and use UML as the human-system interface (e.g., for representing GDMs and transformation rules externally). Tools will be constructed to perform the translation between internal and external representations.

With natural language-like syntax, a TLG specification is self-descriptive and very understandable. Therefore, TLG has more potential to be mastered by software engineers than other formal methods such as Z [Spi89].

XML is very suitable for data exchange and description, but not for code generation or even more complicated tasks like model transformations. In a pure sense, XML carries no more semantic meaning than HTML. XML itself does not perform a computation, but relies on the intelligence of non-reusable XML processing engines. On the other hand, TLG is Turing complete with very nice logic and functional language style reasoning. Regarding readability, the frequent use of angle bracket templates in Xpath and XSLT [Cle01] makes the readability of the generator poor. TLG offers improvements in readability, as well.

TLG is Object-Oriented (OO), making it a good candidate for formal specification of OO computing entities. Additionally, TLG goes beyond OO programming languages with its unique syntax and semantics. A simple rule such as:

```
NewObject:: {Object1}* Object2, Object3; Object4.
```

states a rather complicated feature selection and federated construction of the NewObject: the NewObject could be constructed by zero or more instances from domain Object1 followed by an instance from domain Object2, and an instance from domain Object3; or the NewObject could be constructed by an instance from domain Object4. It would require a large block of statements in an object-oriented programming language to represent the same intent. In TLG, it is very easy to combine objects and flat entities (literals) together as features because both terminals and nonterminals are allowed on the right hand side of meta-rules.

TLG has two levels. The meta level computation can be viewed as model/pattern transformations. More abstract patterns on the left hand side can be substituted by many combinations and alternatives of more specific patterns on the right hand side of the grammar. The hyper level context-free grammar (together with the consistent substitution) sets the context for the first one: rules and logic for applying patterns, very suitable for plug-and-play component composition. Also for each context-free grammar, we can automate the feature configuration validation and constraint checking [Jon02], leveraging widely available open parser and type checker generator facilities such as CUP [CUP99].

6. Conclusion

This extended abstract provides the overview of the UniFrame architecture, considerations in designing UniFrame and the issues of infrastructure implementations. The novel contribution of UniFrame is to bridge the gap between the vision of Generative Programming and the existing

MDA framework: we provide a practical architecture and a infrastructure technique using the MDA model transformation idea to fulfill the goal of Generative Programming.

7. References

[Bry02] B. R. Bryant, B.-S. Lee, "Two-Level Grammar as an Object-Oriented Requirements Specification Language," *Proc. 35th Hawaii Int. Conf. System Sciences*, 2002.

[Cao02] Fei Cao, Barrett Bryant, Carol Burt, Rajeev Raje, Mikhail Auguston, Andrew Olson. "A Translation Approach to Component Specification," (poster), OOPSLA'02,2002.

[Cle01] J. C. Cleaveland. *Program Generators with XML and JAVA*. Prentice Hall 2001.

[Corba] Common Object Request Broker Architecture (CORBA), http://www.corba.org/

[CUP99]CUP parser generator for Java. http://www.cs.princeton.edu/~appel/modern/java/CUP/

[Cza00] Czarnecki, K., Eisenecker, U. W., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[J2SE] Java™ 2 Platform, Standard Edition, http://java.sun.com/docs/index.html

[Jon02] M. D. Jonge, J. Visser "Grammars as Feature Diagrams" *Proceedings of Workshop on Generative Programming*, April 2002. http://www.cwi.nl/events/2002/GP2002/papers/dejonge.pdf

[Kan98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, Moonhang Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Engineering* 5, pp. 143-168, 1998.

[OMG01] Object Management Group. Model Driven Architecture: A Technical Perspective. Technical Report. Document #ormsc/2001-07-01. Framingham, MA: Object Management Group. July 2001.

[OpenG]OpenGL. http://www.opengl.org/

[Raj01] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," *Proc. 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration*, 2001, pp. 109-119.

[Sun02] C. Sun, R. R. Raje, A. M. Olson, B. R. Bryant, M. Auguston, C. C. Burt, Z. Huang, "Composition and Decomposition of Quality of Service Parameters in Distributed Component-Based Systems," to appear in *Proc. Fifth IEEE Int. Conf. Algorithms and Architectures for Parallel Processing*, 2002.

[Spi89] J. M. Spivey, The Z notation: a reference manual. Prentice Hall, New York, 1989.

[UniFr] UniFrame http://www.cs.iupui.edu/uniFrame/

[W3C] World Wide Web Consortium, Web Services, http://www.w3.org/2002/ws/

[Zha02] W. Zhao. "Two-Level Grammar as the Formalism for Middleware Generation in Internet Component Broker Organizations." *Proceedings of GCSE/SAIG Young Researchers Workshop*, held in conjunction with the *First ACM SIGPLAN Conference on Generative Programming and Component Engineering*, 2002. http://www.cs.uni-essen.de/dawis/conferences/GCSE_SAIG_YRW2002/submissions/final/Zhao.pdf

Some Axioms and Issues in the UFO Dynamic Analysis Framework

Clinton Jeffery
Department of Computer Science
New Mexico State University
jeffery@cs.nmsu.edu

Mikhail Auguston
Department of Computer Science
Naval Postgraduate School
maugusto@nps.navy.mil

Abstract

UFO is a framework for constructing dynamic analysis tools that require varying degrees of access and control over program executions. UFO combines run time and post-mortem techniques to perform required analyses. Declarative and imperative notations are provided for constructing monitors at appropriate semantic levels. Multiple analyses can be bundled into a given monitor, and multiple monitors can be applied to a given target program execution. This paper presents the central tenets of UFO, along with our current set of research challenges.

1. Motivation

Automatic debugging and program visualization are two of the most promising application areas of dynamic analysis, with potential to impact on crucial areas of software development and maintenance. We believe the slow rate of advancement in these areas is due to the high cost of developing new tools. We have previously focused on a language (FORMAN) and an architecture (Alamo) that reduce these costs [1][2][4]. FORMAN is a special-purpose language for expressing dynamic analyses; it has been implemented previously for subsets of Pascal and C. Alamo is a lightweight architecture for program execution monitoring; it has been implemented for a subset of C and for the virtual machine used by the Icon and Unicon programming languages. The virtual machine implementation of Alamo is attractive for research because it provides high performance and superior ease of use for a full-size "real" programming language, allowing testing on large programs and the possibility of deploying successful tools to a user community.

We recently merged the FORMAN and Alamo efforts to produce UFO (Unicon-FORMAN), a framework for rapidly constructing dynamic analyzers [3][4]. We have used UFO to construct a variety of simple automatic debuggers and visualization tools that run well on small and medium sized applications. Our next efforts must walk the tightrope of scaling up to production tools for large applications, while retaining the power and ease of use that are characteristic of the current research UFO system. With that in mind, this paper presents the central tenets of the UFO system, and concludes with an exploration of the current research problems and our plans to address them.

2. Axioms

UFO is primarily an implementation of FORMAN built on top of the Alamo monitor architecture. Early experiments showed the marriage to improve FORMAN speed by two orders of magnitude and shorten the lines of code necessary to write Alamo monitors by one order of magnitude. This section sketches the primary characteristics of UFO.

- A precise program behavior model, in which semantics of the monitored language are mapped to directed acyclic graphs of events. These graphs are defined using an *event grammar*, a notation that approximates the semantics of the language to be monitored. The behavior model is essential to provide general purpose capabilities for a wide range of tools.
- A declarative special-purpose monitoring language, tailored specifically for dynamic analyses expressed in terms of patterns within the graphs of events. This component is necessary to reduce the cost of developing new tools. Section 4 provides some examples; shorthand refinements to improve the syntax could be explored after the main semantics and performance issues are resolved.
- An hybrid execution model, in which most analysis work is performed at run-time, and more complex analyses transparently combine run-time collection and partial analysis with more extensive post-mortem analysis. This element is necessary but not sufficient by itself to achieve acceptably high performance for large scale production systems. This important element is new in UFO, compared with previous FORMAN and Alamo efforts. It provides high performance.
- Automatic instrumentation provided by special-purpose virtual machine support; static or dynamic configuration of VM instrumentation; no recompilation, relinking, or alteration of target program executables to be monitored. This provides substantial ease of use.

3. Some Research Issues and Challenges

UFO's chief design goals revolve around notational power and ease of use. The current prototype implementation of UFO [5][5] processes millions of events per minute. But, for large programs higher performance is needed. This goal motivates several open problems we are pursuing.

Minimizing the number of context switches. UFO's run-time execution model is based on lightweight coroutine switches between monitors and the program being observed. This separation is a compromise between intrusive in-line single-thread execution used in low-cost analysis tools such as profilers, and the complete separation imposed by high-cost analysis tools such as debuggers. One research goal is to retain the abstraction and low-intrusion benefits of the coroutine model without having to pay (so much) for it.

Virtual machine configuration and customization. The VM instrumentation can be turned off at multiple levels, including compile-time via #ifdef and run-time via a dynamic filter that controls whether instrumented or uninstrumented versions of functions are called, and whether an event report (via lightweight context switch) is performed for a given instrumentation site. This configuration can be further exploited by having the UFO compiler generate a custom VM with exactly the instrumentation it needs for a particular monitoring application. The central VM interpreter function (interp()) can benefit from a finer granularity of customization than the current instrumented-versus-uninstrumented options; it is critical to performance and contains 30 of the 119 types of events instrumented in the VM. Generating a custom VM may greatly improve monitoring performance within this VM interpreter loop. The VM generation system needs to make it easy and convenient for the UFO compiler to generate custom VM's and associate them with generated analyzers in a persistent manner. Custom VM's should be shareable by monitors that use the same events.

Inter-monitor optimizations. When multiple analyses are compiled together, substantial cost savings might be obtained by factoring common tasks such as event data collection. For example, a profiler that computes summaries and a visualizer that shows run-time details might operate on the same information, and might even share some common analysis structures.

Meta-events and analysis hierarchies. UFO's event model composes higher level events from lower level ones, but analysis tools create additional information which may constitute the input for higher level analyses. This facilitates the sharing of analysis information among tools, reducing the cost of running multiple tools.

4. Examples of debugging rules

Alamo's goal was to reduce the difficulty of writing execution monitors to be just as easy as writing other types of application programs. UFO supports FORMAN's more ambitious goal of reducing the difficulty of writing automatic debuggers to the task of specifying generic assertions about program behavior.

This section presents formalizations of typical debugging rules. UFO supports traditional precondition checking, or print statement insertion, without any modification of the target program source code. This is especially valuable when the precondition check or print statement is needed in many locations scattered throughout the code.

Example #1: Tracing. Probably the most common debugging method is to insert output statements to generate trace files, log files, and so forth. It is possible to request evaluation of arbitrary Unicon expressions at the beginning or at the end of events. The virtual machine evaluates these expressions at the indicated time moments.

```
FOREACH  A: func_call &
        A.func_name == "my_func"
      FROM prog_ex
  A.value_at_begin(
     write("entering my_func, value of X is:", X) ) AND
  A.value_at_end(
    write("leaving my_func, value of X is:", X) )
```

This debugging rule causes calls to write() to be evaluated at selected points at run time, just before and after each occurrence of event A.

Example #2: Profiling. A myriad of tools are based on a premise of accumulating the number of times a behavior occurs, or the amount of time spent in a particular activity or section of code. The following debugging rule illustrates such computations over the event trace.

```
SAY( "Total number of read() statements: "
     CARD[ r: input & r.filename == "xx.in"
            FROM prog_ex ]
      "Elapsed time for read operations is: "
      SUM [ r: input & r.filename == "xx.in"
             FROM prog_ex   APPLY r.duration] )
```

Example #3: Pre- and Post- Conditions. Typical use of assertions includes checking pre- and post-conditions of function calls.

```
FOREACH A:func_call & A.func_name=="sqrt"
      FROM prog_ex
  A.paramlist[1] >=0 AND
  abs(A.value*A.value-A.paramlist[1]) < epsilon
WHEN FAILS SAY("bad sqrt(" A.paramlist[1]
                         ") yields " A.value)
```

4.1 Generic Bug Descriptions

Another prospect is the development of a suite of generic automated debugging tools that can be used on any Unicon program. UFO provides a level of abstraction sufficient for specifying typical bugs and debugging rules.

Example #4: Detecting Use of Un-initialized Variables. Reading an un-initialized variable is permissible in Unicon, but often leads to errors. In this debugging rule all variables in the target program are checked to ensure that they are initialized before they are used.

```
FOREACH V: variable  FROM prog_ex
          FIND D: lhp FROM V.prev_path
                   D.source_text == V.source_text
    WHEN FAILS SAY( " uninitialized variable "
                               V.source_text)
```

Example #5: Empty Pops. Removing an element from an empty list is typical of expressions that fail silently in Unicon. While this can be convenient, it can also be a source of difficult to detect logic errors. This assertion assures that items are not removed from empty lists.

```
FOREACH   a: func_call &
       a.func_name == "pop" AND
       a.value_at_begin( *a.paramlist[1] == 0)
    SAY("Popping from empty list at event " a)
```

5. Implementation Issues

The most important of these issues is the translation model by which FORMAN assertions are compiled down to Unicon Alamo monitors. Debugging activities are written as if they have the complete post-mortem event trace, the DAG with events, event attributes, and precedence and containment relations, available for processing. This generality is extremely powerful; however, for most practical uses we have seen, assertions can be compiled down into monitors that execute entirely at runtime. Runtime monitoring saves enormously on memory and I/O requirements and is the key to practical implementation. For those assertions that require post-mortem analysis, the UFO runtime system computes a projection of the execution DAG necessary to perform the analysis.
The UFO compiler generates Alamo Unicon monitors from FORMAN rules. Each FORMAN statement is translated into a combination of initialization, run-time, and post-mortem code. Monitors are executed as coroutines with the Unicon target program.

Monitors generated by the UFO compiler reduce complex assertions to the single event loop. Keeping event detection in a single loop allows uniform processing of multiple event types used by multiple monitors. The code generated by the UFO compiler integrates event detection, attribute collection, and aggregate operation accumulation in the main event loop.

Assertions in UFO may use nested quantifiers implying two nested loops, so code generation addresses this issue by flattening the main loop structure, and postponing assertion processing until required information is available. An hybrid code generation strategy performs runtime processing whenever possible, delaying analyses until post-mortem time when necessary. Different assertions require different degrees of trace projection storage; code responsible for trace projection collection is also arranged within the main loop. The following generation template gives a flavor of the UFO trace projection mechanism.

Rules with two nested quantifiers of the form

```
Quantifier A: Pattern_A
     Quantifier B: Pattern_B FROM A
             Body
```

utilize a monitor whose main loop follows the pattern:

```
Main Loop
     Maintain stack of nested A events
     Accumulate events B in a B-list
     If end of event A
       Loop over B-list
             Do Body
     Endif
     If stack of A is empty
             Destroy B-list
End of Main Loop
```

This requires accumulation of a trace projection for B-events and may cause a mild overhead at the run time.

5.1 Optimization Issues

The UFO approach combines an optimizing compiler for monitoring code with efficient run-time event detection and reporting. Since we know at compile time all necessary event types and attributes required for a given UFO rule, the generated Unicon monitor can be very selective about the behavior that it observes.

For certain kinds of UFO constructs, such as nested quantifiers, the monitor must accumulate a sizable projection of the complete event trace and postpone corresponding computations until all required information is available. The presence of the previous_path and following_path attributes in UFO rules triggers this kind of optimization; previous_path and following_path are used in rules which specify preceding or following contexts for events of interest.

For further optimization, especially in the case of programs containing a significant number of modules, the following FORMAN construct limits event processing to events generated within the bodies of functions F1, F2, ... , Fn.

```
WITHIN F1, F2, ... , Fn DO
     Rules
END_WITHIN
```

This provides for monitoring only selected segments of the event trace.

Unicon expressions included in the value_at_begin and value_at_end attributes are evaluated at run time.

Some other optimizations implemented in this version are:

- only attributes explicitly used in the UFO rule are collected in the generated monitor;
- an efficient mechanism for event trace projection management, which disposes from the stored trace projection those events that are no longer used after a certain rule has been fully evaluated;
- both event types and context conditions are used to filter events for the processing.

UFO's goal of practical application to real-sized programs has motivated several improvements to the already carefully-tuned Alamo instrumentation of the Unicon virtual machine. We are working on additional optimizations.

We expect that the most promising optimizations are within the generation of instances of Virtual Machine tailored for a particular monitoring task.

6. Conclusions

The architecture employed in UFO could be adapted for a broad class of languages such as those supported by the Java VM or the .net VM. Our approach to dynamic analysis uniformly represents many types of debugging-related activities as computations over traces, including assertion checking, profiling and performance measurements, and the detection of typical errors. We have integrated event trace computations into a monitoring architecture based on a virtual machine. Preliminary experiments demonstrate that this architecture is scalable to real-world programs.

One of our next steps is to build a repository of formalized knowledge about typical bugs in the form of UFO rules, and gather experience by applying this collection of assertions to additional real-world applications. There remain many optimizations that can improve the monitor code generated by the UFO compiler; for example, merging common code used by multiple assertions in a single monitor, and generating specialized VMs adjusted to the generated monitor.

Acknowledgements

This work has been supported in part by U.S. Office of Naval Research Grant # N00014-01-1-0746, by U.S. Army Research Office Grant # 40473--MA-SP, and by the National Library of Medicine.

References

[1] M. Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, in the Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, AADEBUG'95, Saint-Malo, France, May 22-24, 1995, pp. 277-291.

[2] Clinton L. Jeffery, Program Monitoring and Visualization: an Exploratory Approach. Springer, New York, 1999.

[3] M. Auguston, A. Gates, M. Lujan, "Defining a Program Behavior Model for Dynamic Analyzers", in the Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97, Madrid, Spain, June 1997, pp. 257-262.

[4] M. Auguston, "Lightweight semantics models for program testing and debugging automation", in Proceedings of the 7th Monterey Workshop on "Modeling Software System Structures in a Fast Moving Scenario", Santa Margherita Ligure, Italy, June 13-16, 2000, pp.23-31.

[5] M. Auguston, C. Jeffery, and S. Underwood. "A Framework for Automatic Debugging", IEEE 17th Intl. Conf. on Automated Software Engineering, Edinburgh, September 2002, IEEE Computer Society Press, pp.217-222

[6] C. Jeffery and M. Auguston. "Towards Fully Automatic Execution Monitoring". Monterey Workshop 2002, Venice, October 2002, sponsored by US Army Research Office and NSF, pp.232-243

[7] Clinton Jeffery, Shamim Mohamed, Ray Pereda, and Robert Parlett, "Programming with Unicon", http://unicon.sourceforge.net.

[8] Ralph E. Griswold and Madge T. Griswold, The Icon Programming Language, 3rd edition. Peer to Peer Communications, San Jose, 1997.

Automating Feature-Oriented Domain Analysis

Fei Cao, Barrett R. Bryant, Carol C. Burt
Department of Computer and Information Sciences
University of Alabama at Birmingham
{caof, bryant, cburt}@cis.uab.edu

Zhisheng Huang, Rajeev R. Raje, Andrew M. Olson
Department of Computer and Information Science
Indiana University Purdue University at Indianapolis
{zhuang, rraje, aolson}@cs.iupui.edu

Mikhail Auguston
Computer Science Department
Naval Postgraduate School
auguston@cs.nps.navy.mil

Abstract

Feature modeling is commonly used to capture the commonalities and variabilities of systems in a domain during Domain Analysis. The output of feature modeling will be some reusable assets (components, patterns, domain-specific language, etc.) to be fed into the application engineering phase for ultimate software products. But current practice lacks an automatic approach for seamless generation of reusable assets from feature models. This paper presents an algorithm for generating sets of instance descriptions (feature instances) from feature models of a domain and applies this algorithm in creating a Generic Feature Modeling Environment for automating Feature-Oriented Domain Analysis.

Keywords: Feature Modeling, Domain Analysis, Generative Programming

1. Introduction

Generative Programming (GP) [Czar00] has emerged as a software development paradigm for automatic generation of software products based on modeling of software system families. The distinct property of GP is it is not only about a development *for reuse* in terms of building a Generative Domain Model (GDM) for software system families, but also about a development *with reuse* in terms of using GDM to generate concrete systems. To build a GDM, domain analysis has to be applied to scope a system family and to identify the commonalities, variabilities and dependencies among family members. A crucial outcome of the domain analysis phase is a feature model, which is usually represented as a feature diagram. However, the application of feature diagrams is quite limited, due to the fact that current practice is not fully automated, while the size of the set of feature instances may be expanded exponentially (which we will see later in this paper), thus it is difficult to apply constraint checking and other types of computing. In order to align with the goal of GP for the highest level of automation, to cope with family system processing (which is usually of a large scale), feature modeling should be carried out in an automatic fashion to seamlessly generate reusable assets to be used in application engineering for constructing a family of applications. This paper presents an algorithm for generating the set of all feature instances from a feature diagram and applies this algorithm in creating a Generic Feature Modeling Environment (GFME) for automating Feature-Oriented Domain Analysis (FODA). This paper is organized as follows: Section 2 briefly describes major related research efforts. Section 3 gives the algorithm for computing feature models. Section 4 presents the GFME created with the Generic Modeling Environment (GME) 2000 [GME01]. Section 5 draws the conclusion of this paper.

2. Related Work

Feature models were initially introduced by the FODA method [Kang90]. In the FODA method, a feature is defined as an end-user-visible characteristic of a system. This model uses a feature diagram to represent a hierarchical decomposition of features, which include mandatory, alternative or optional features. Feature constraints, stakeholders and rationales are also incorporated in this feature model. Czarnecki and Eisenecker [Czar00] give a more detailed account of feature diagrams including diagram normalization.

The FODA method uses Prolog in a prototype tool for doing checking over some sets of feature values. However, features have to be stored in the Prolog fact base first, rather than being analyzed directly over the feature diagram, thus the tool is not seamlessly integrated with the visual diagram setting. Czarnecki and Eisenecker [Czar00] also explore the possible implementation of feature diagrams by mapping into UML, which in turn may be used to generate some implementation codes using such CASE tools as Rational Rose[1]. The mapping process, however, is again a manual process. Also, what Rational Rose can generate are just some skeleton codes, which are far from being complete implementations.

Feature models can be represented not only in graphical form using feature diagrams, but also in textual form. Van Deursen and Klint [Deur02] propose a Feature Description Language (FDL) for textual representation of feature diagrams. Manipulation of features is achieved by Feature Diagram Algebra (FDA), which consists of four sets of rules: normalization rules, variability rules, expansion rules and satisfaction rules. The FDL can be fed into the a tool named "ASF+SDF Meta-Environment" [Bran01] for direct execution as a basis for prototype tool support, which again is not seamlessly integrated with graphical representations of feature diagrams; the capacity of constraint checking is quite limited; the FDA is separated from, rather than integrated as part of the feature diagram; the generation of reusable assets from FDL is not flexible.

Obviously for the related work mentioned in this section, there is a gap between using feature diagrams for feature modeling and a seamless, efficient generation of reusable assets. This paper presents an approach toward bridging this gap.

3. An Algorithm for Feature Diagram Computing

In contrast to computing features by transforming feature diagrams to some other representation forms (such as UML or FDL) first, we are going to apply the proposed algorithm directly over the feature diagram. We first briefly describe the representations used in [Czar00] illustrated in Figure 1. The *mandatory* feature is represented by being attached to an edge ending with a filled circle. So the feature F consists of both C1 and C2 in this case, and the feature instances here are {F, C1, C2}. The *optional* feature is represented by being attached to an edge ending with an unfilled circle. So the feature F may or may not contain C1. The *optional* feature instances here are {F, C2} and {F, C1, C2}. The *alternative* feature is represented by connecting edges with an arch. So the feature F consists of exactly one of its child features. The *alternative* feature instances here are {F, C1} and {F, C2}. Note that if C1 is optional while C2 is mandatory, then the *alternative* feature instances here are {F}, {F, C1} and {F, C2}, because the child feature instances derived from the C1 side contain an empty feature. The *Or* feature is represented by connecting edges with a filled arch. The *Or* feature instances here are {F, C1}, {F, C2} and {F, C1, C2}. If there is an optional child feature, then the *Or* representation is actually equivalent to the situation that all the child features are optional, i.e., the *Or* feature instances will be {F}, {F, C1}, {F, C2} and {F, C1, C2}.

These representations can also be intermingled in feature diagrams, such as in Figure 2. These mixture forms can be normalized so that it is easier to be processed. e.g., Figure 2 can be normalized into Figure 3.

This normalization can be performed iteratively over all such "mixture relation" nodes in the feature diagram. In this way, the father-feature in the feature diagram will only be either *XOR* (corresponding to *alternative*), or *OR*, or *AND* in relationship to child-features. Meanwhile, each child-feature may be either *optional* or *mandatory*. Obviously, the normalization process described here is fulfilled by adding hierarchy into the original feature tree

[1] www.rational.com

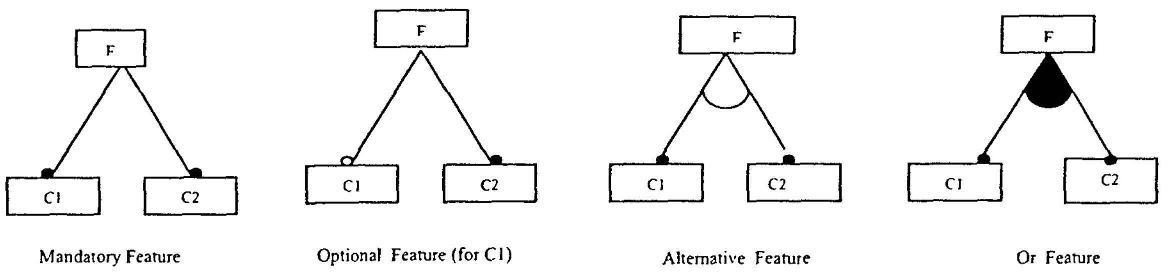


Figure 1: Feature Diagram Representation

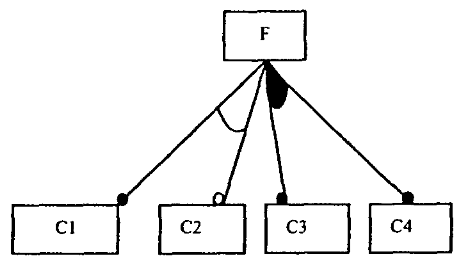


Figure 2: Mixture of Feature Representation

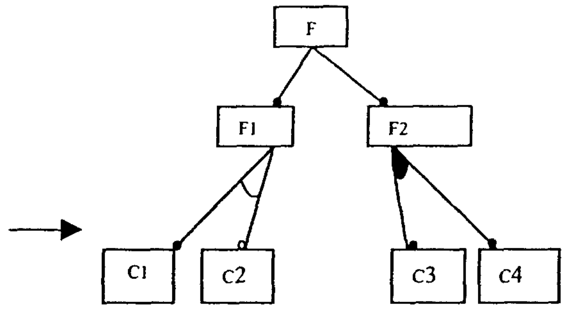


Figure 3: Normalized Feature Representation

without loss of any commonality and variability representations. After such normalization is performed, the feature diagram will be in the structure as in Figure 4. The proposed algorithm will be applied over such normalized feature diagrams thereafter.

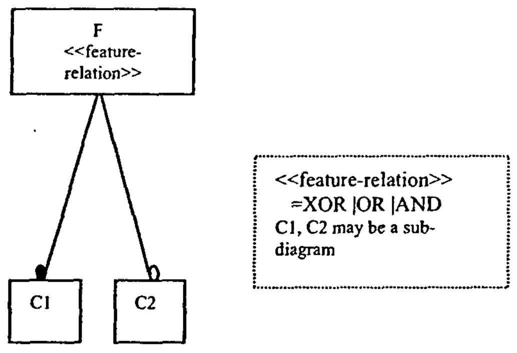


Figure 4: Variation of Feature Diagram

Suppose each feature node is represented as the following data structure (note that without loss of generality, the following data structure may not be strictly consistent with a specific C++ programming environment):

```
struct  FeatureNode{
 String featurename;
```

```
enum {XOR, OR, AND} feature-relation;
  /*denotes the father-child  relation */

 ChildConnectionList *edges;
  /*list of connections associated with
   its child-feature nodes */
  }

 struct ChildConnectionList {
   bool  isMandatory ;
     /*is a mandatory/optional  feature*/
   FeatureNode * aFeature;
        /*point to a feature node*/
                             }
```

From the data structure above we can see that we can get access to the child-nodes of a feature node by traversing its associated edges.

Currently, the result of the algorithm to compute the feature diagram is just the set of all feature instances of a feature diagram. The result will be represented as a list. Each element of the list corresponds to a feature instance. Each feature instance in turn is represented as a list, which consists of the list of pointers to the related feature node. The result is represented as follows:

```
typedef List<FeatureInstance *> Result;
typedef List<FeatureNode *>
                 FeatureInstance;
```

Below is the pseudo code for the algorithm. The input parameter to the algorithm is the pointer to the root node of a feature diagram. The output will be all feature instances derived from the feature diagram. Note the variables are in italicized font while the types are in bold font.

```
Result * processFeatureDiagram (
   FeatureNode *node-root)
 {
 create a temp1:FeatureInstance  with
 only node-root in it;

 create a temp2: Result  with only one
 FeatureInstance temp1 in it;

 if(node-root has no child nodes)
 then  return temp2;

 else
 if (node-root->feature-relation==AND)

  {
  recursively call  processFeatureDiagram
  over each of node-root's child-nodes,
  each returning a child result;

  if corresponding child node is
  "Optional",
  add an empty FeatureInstance into the
  corresponding child result;

  calculate the production of all the
  returned child results as temp3:Result;

  return the production of temp2 and
  temp3;
  }

 else
 if(node-root->feature-relation==XOR)
  {
  recursively call processFeatureDiagram
  over each of node-root's child-nodes,
  each returning a child result;

  calculate the union of those returned
  child results as temp3:Result;

  if there is a child node that is
  "Optional",
  add an empty FeatureInstance into
  temp3;

  return  temp3;
  }

 else
 if(node-root->feature-relation==OR)
  {
   recursively call processFeatureDiagram
   over each of node-root's child-nodes,
   each returning a child result;

   for each of the child result returned
   in the above call,
   add an empty FeatureInstance into it;

   get the production of all the child
   results as temp3:Result;

   If all child features are mandatory,
   remove the empty FeatureInstance from
   temp3;

   return the production of temp2 and
   temp3;
   }
 }
```

Beware that a **Result** is actually a two-dimension data structure. If **Result** *A* has m **FeatureInstances** while Result *B* has n **FeatureInstances**, then the union of *A* and *B* has m+n **FeatureInstances** while the production of *A* and *B* has m*n **FeatureInstances**. To exemplify the above algorithm, we use ε to represent an empty **Result**, × for production, ∪ for union operation in Figures 5-7, which correspond to three types of cases for computing the set of feature instances. Also from Figure 7 we can easily see the size of feature set may grow exponentially (as to the extreme case where all *feature-relation*s are *OR* , the size will be 2^n, where n is the amount of leaf nodes).

Here we put the non-leaf node (like *F* here) into the feature instances in order to facilitate constraint checking. If one non-leaf feature *F* is supposed to be excluded in the final feature instance, then its child-features should not be included correspondingly, and we can eliminate those feature instances from the final result by identifying which feature instance contains feature *F*, rather than by tracking down all its child-features laboriously.

4. A Generic Feature Modeling Environment (GFME)

We use the Generic Modeling Environment (GME) [GME01] to build GFME. GME is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The configuration is accomplished through metamodels specifying the modeling paradigm (modeling language) of the application domain. The modeling paradigm defines the family of models that can be created using the resultant modeling environment. The metamodels specifying the modeling paradigm are used to automatically generate the target domain-specific environment. GME provides the Builder Object Network (BON) framework for building interpreters to interpret domain models built in the domain-specific environment. The interpretation process can be used to generate reusable assets for the domain engineering phase. The BON API provides leverages for access to the domain models, which makes the above algorithm implementable. With all those facilities of GME, we believe it has the best tool support for feature modeling.

GFME provides the modeling environment for building feature diagrams with the structure as described in Figure 4. Figure 8 provides the screenshot of the GFME. Note at the lower-right corner is the interface to specify such attributes as the relationship with its child-nodes for a node under focus (here "TransactionSubsystem") in the environment. In the same way, we can specify the attributes for those connections

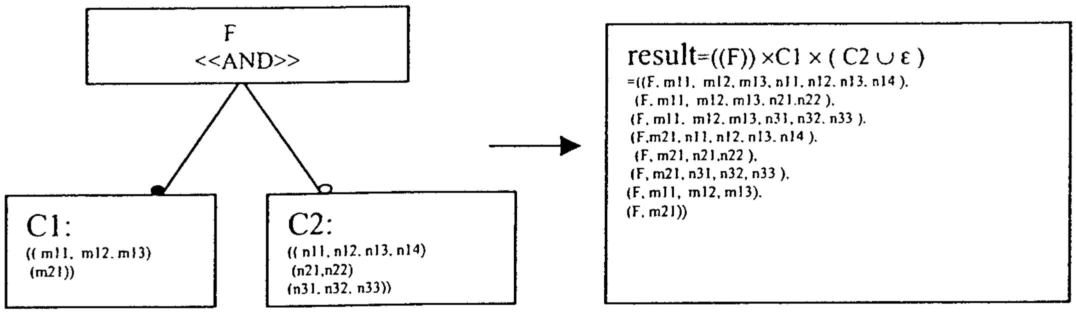

F
<<AND>>
C1:
((m11, m12, m13)
(m21))
C2:
((n11, n12, n13, n14)
(n21,n22)
(n31, n32, n33))
result=((F)) ×C1 × (C2 ∪ ε)
=((F, m11, m12, m13, n11, n12, n13, n14),
(F, m11, m12, m13, n21,n22),
(F, m11, m12, m13, n31, n32, n33),
(F,m21, n11, n12, n13, n14),
(F, m21, n21,n22),
(F, m21, n31, n32, n33),
(F, m11, m12, m13),
(F, m21))

Figure 5: Computing AND result

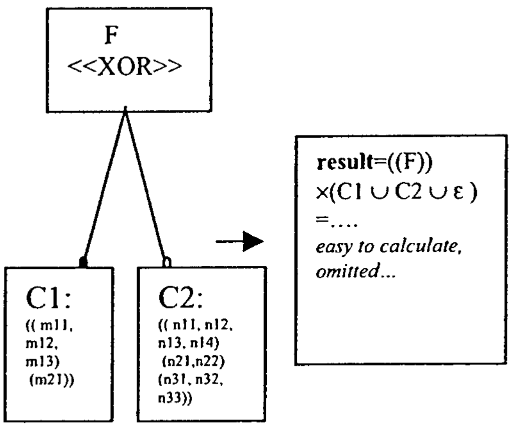


Figure 6: Computing XOR result

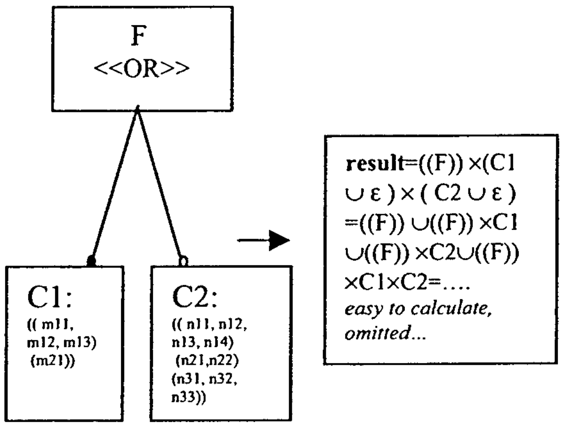

F
<<OR>>
C1:
((m11, m12, m13)
(m21))
C2:
((n11, n12, n13, n14)
(n21,n22)
(n31, n32, n33))
result=((F)) ×(C1 ∪ ε) × (C2 ∪ ε)
=((F)) ∪((F)) ×C1 ∪((F)) ×C2∪((F)) ×C1×C2=....
easy to calculate, omitted...

Figure 7: Computing OR result

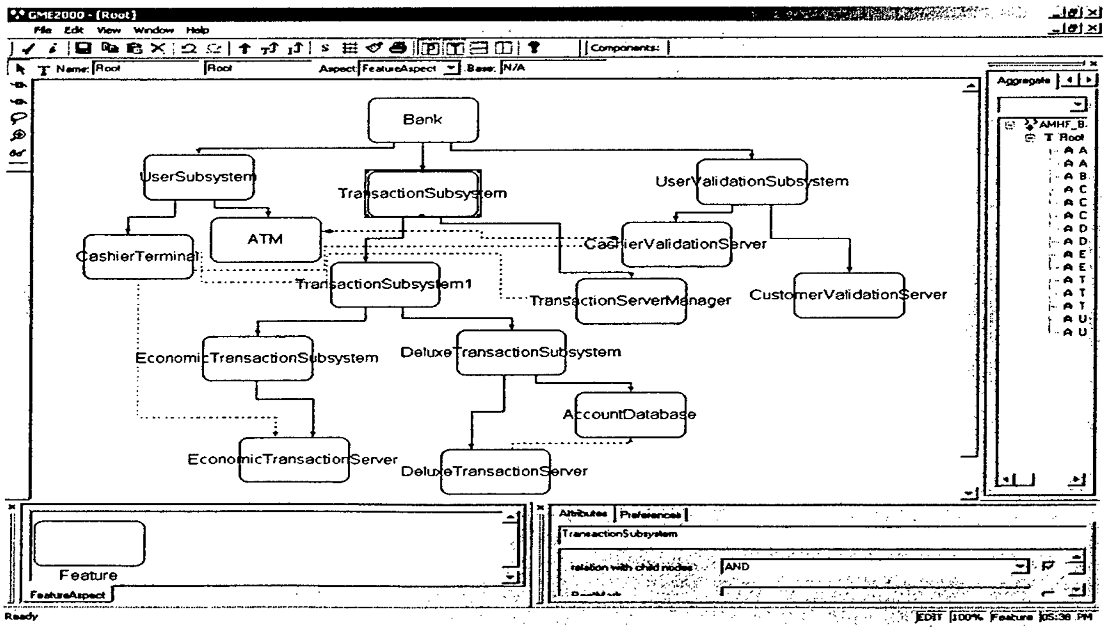


Figure 8: Generic Feature Modeling Environment

between feature nodes. The dashed lines denote the various kinds of dependencies or constraints to be enforced between feature nodes. Currently we just generate the set of feature instances from feature diagram satisfying all specified constraints. With full control of the interpretation process (i.e., writing interpreter code via BON API), we can generate application code from feature diagrams on demand.

5. Conclusion

Feature Modeling is the core part of FODA. Our ongoing UniFrame project [Raje02] requires feature modeling for building a generative domain model. The reusable assets generated from feature modeling after normalization, expansion and constraint checking will be output into XML files. The reusable assets serve two purposes: 1) for clients to initiate natural-language-like queries [Lee02] in the problem space [Czar00]; 2) to provide a guideline for component providers to produce component families in the solution space [Czar00]. The current practice of feature modeling remains at the manual or semi-automatic level, which hinders it from becoming widely applied. This paper applies normalization over the traditional feature diagram and presents an algorithm to generate complete feature instances from a feature diagram under constraints. The algorithm is adopted in GFME, which provides an efficient, automatic FODA environment.

Acknowledgements. This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

References

[Bran01] M.G.J. van den Brand, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, J. Visser. *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment.* Compiler Construction (CC '01), vol. 2027, Lecture Notes in Computer Science, pp. 365-370, Springer-Verlag, 2001.

[Czar00] K. Czarnecki, U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, 2000.

[Deur02] A. van Deursen and P. Klint. *Domain-specific Language Design Requires Feature Descriptions.* Journal of Computing and Information Technology 10(1), pp. 1-17, 2002.

[GME01] *GME 2000 User's Manual, Version 2.0.* ISIS, Vanderbilt University, 2001.

[Kang90] K.C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-oriented Domain Analysis (FODA) Feasibility Study.* Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[Lee02] B.-S. Lee, B. R. Bryant. *Contextual Processing and DAML for Understanding Software Requirements Specifications.* Proceedings of COLING 2002, the 19th International Conference on Computational Linguistics, pp. 516-522, 2002.

[Raje02] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt. *A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components.* Concurrency and Computation: Practice and Experience 14, pp. 1009-1034, 2002.

Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control *

Carol C. Burt
Barrett R. Bryant
University of Alabama Birmingham
cburt, bryant@cis.uab.edu

Rajeev R. Raje
Andrew Olson
Indiana University Purdue University Indianapolis
rraje,aolson@cs.iupui.edu

Mikhail Auguston
Naval Post Graduate School
auguston@cs.nps.navy.mil

Abstract

The research vision of the Unified Component Meta Model Framework (UniFrame) is to develop an infrastructure for components that enables a plug and play component environment where the security contracts are a part of the component description and the security aware middleware is generated by the component integration toolkits. That is, the components providers will define security contracts in addition to the functional contracts. These security contracts will be used to analyze the ability of a service to meet the security constraints when used in a composition of components. A difficulty in progressing the security related aspects of this infrastructure is the lack of a unified access control model that can be leveraged to identify protected resources and access control points at the model level. Existing component technologies utilize various mechanisms for specifying security constraints. This paper will explore issues related to expressing access control requirements of components and the resources they manage. It proposes a platform independent model (PIM) for the access control that can be leveraged to parameterize domain models. It also outlines the analysis necessary to progress a standard transformation from this PIM to three existing Platform Specific Models (PSMs).

1. Introduction

Enterprises are increasingly dependent upon multiple middleware technologies that enable new business paradigms by weaving together legacy systems with advanced technology. Component-based system integration supports core business functionality, integrates business processes and enables companies to communicate with customers, suppliers, and business partners. The Unified Component Meta-Model Framework (UniFrame) [1] attempts to unify distributed component models under a common meta-model for the purpose of enabling the discovery, interoperability, and collaboration of components via generative software techniques. This research targets the dynamic assembly of distributed software systems from components developed using different component models, and explores how the quality of service (QoS) requirements, such as security, influence the design of components and their compositions. The inherent complexity of such integrations introduces significant challenges for controlling access to application resources (business, customer and personal information as well as product and application features). This paper expands on our previous work [2, 3] to explore how Model Driven Architecture techniques may be used for an integration of the access control solutions in heterogeneous environments.

OMG's Model Driven Architecture (MDA) [4] initiative facilitates the standardization of Platform Independent Models (PIMs) and the transformation of those models to multiple Platform Specific Models for implementation. One of the challenges of Model Driven Architecture is the existence of Platform Specific Models that do not adhere to a unified Platform Independent Model. In such cases, bridging is, at best, hand crafted and at worse, impossible. Today this is the case for the access control models. What is needed is a Platform Independent Model for access control that forms the foundation of end-to-end access controls in heterogeneous computing environments. This model must accommodate existing Platform Specific Models while providing the flexibility for innovation in access control technology.

This paper proposes a Platform Independent Model for access control (AC-PIM) that provides a clear architectural separation between the access policy (the management and expression of access rules), the access

* This research was supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

decision (evaluating policy at a given point in time), and the access control (the enforcement of access decisions). The paper also explores access control models adopted via different standards organizations and outlines the transformations to their access control Platform Specific Models.

2. Relevant Research and Standards

The ITU-T recommendation X.812 (ISO/IEC 10181-3) [5] provides a reference model for the access control that is consistent with the model proposed in this paper. There are also several consortium and de facto standards that are relevant for this work. They are outlined below. The detailed work will select three of these models for analysis.

2.1 Globus GRID Research

Globus [6] is a research project that focuses not only on the issues associated with the building of computational grid infrastructures, but also on the problems that arise in designing and developing applications that use grid services. Globus is developing basic security algorithms for secure group communications, management of trust relationships, and developing new mechanisms for fine-grained access control. Globus authorization requirements and the issues that arise with current authorization technologies in GRID are outlined in [7].

2.2 OASIS

OASIS [8] is a not-for-profit global consortium that drives the development, convergence and adoption of e-business standards. There are two OASIS specifications that are of interest. They are the eXtensible Access Control Markup Language (XACML) [9] and the Security Assertion Markup Language (SAML) [10]. XACML is an XML specification for expressing policies for information access over the Internet. SAML is an XML-based security standard for exchanging the authentication and authorization information.

2.3 Object Management Group (OMG)

The Object Management Group (OMG) [11] is an open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications. The OMG Resource Access Decision Facility (RAD) [12] provides a uniform way for application systems to implement a fine-grain access control where the protected resources may be physical, logical, or conceptual or understood only within the context of the business application.

2.3 Java and the Java Community Process

Java provides a Java™ Authentication and Authorization Service (JAAS) [13] package that enables services to authenticate and enforce access controls upon users. J2EE and Java Connectors are required to utilize this model. The Java Community Process (JCP) [14] is an open organization of international Java developers and licensees whose charter is to develop and revise Java technology specifications, reference implementations, and technology compatibility kits. Java Specification Request 115 (JSR-115) [15] is progressing a Java Authorization Contract for Containers (JACC). The Java Authorization Contract for Containers (JACC) seeks to define a contract between containers and authorization service providers that will result in the implementation of providers for use by containers.

2.4 Microsoft ASP.NET

ASP.NET [16] supports the traditional methods of performing the access control (file based) and also provides an URL authorization, which allows administrators to provide an XML configuration that allows or denies an access to URLs based on the current user or the role [17]. Developers can explicitly code authorization checks in their application or can take advantage of the common language runtime's support for declarative security. ASP.NET offers an extensible security architecture that allows the developer to write a custom authentication or authorization server.

2.5 NIST Role-Based Access Control

The National Institute of Standards has proposed a voluntary consensus standard for the Role-Based Access Control [18]. The role based access control (RBAC) is a technology that is attracting an increasing attention, particularly for the commercial applications, because of its potential for reducing the complexity and cost of security administration in large networked applications. Since the publication of the Ferraiolo-Kuhn model [19] for RBAC in 1992, most information technology vendors have incorporated RBAC into their product line, and the technology is finding applications in areas ranging from health care to defense, in addition to the mainstream commerce systems for which it was designed. The RBAC has become the de facto standard for access control in component and web environments. This work is the result of the significant NIST research and patents that they hold on the access control technologies [20]. It is our goal for the access control platform independent model to accommodate (but not require) the NIST RBAC model.

3. Access Control Unification Issues

The first step toward Model Driven Architectures that include access control parameterization and/or authorization contracts is the establishment of a common vocabulary. Although generalized frameworks for the access control have established a common vocabulary for operating system enforced access control models [5, 21], there is no standard vocabulary (or several depending on the perspective!) for discussing the modeling elements of the access control across heterogeneous distributed computing and component-based platforms. This is immediately evident after examining the "standards" that have been progressed to enable a fine-grain access control in these platforms. For example, the OMG Resource Access Decision Facility has an "AccessDecision" object (ADO) that provides the "access decisions" based on the "security attributes of a principal", a "named resource" and an "operation" on the resource [22]. The OASIS SAML/XACML access control model defines an "AuthorizationAuthority" which provides the "authorization decisions" based on "attributes of a subject" and an "action" [9, 10]. The Java2 Enterprise Edition (J2EE) model defines a SecurityManager which enforces the access controls and consults with an AccessController that provides the access decisions based on the permissions granted to a Principal (in native Java this is the same model except the permissions are granted to a Codebase) [13].

Fortunately these models contain many architecturally consistent elements; for example, an AccessDecision object, an AuthorizationAuthority, and an AccessController represent the same architectural element in access control architecture. They do not share a common reference model, however, so it is difficult to determine without a significant analysis if they provide equivalent semantics or not. The interface definition languages are also different. The OMG standardizes the data formats and interfaces for requesting access decisions via ISO IDL [23]; the OASIS specification uses XML schemas [24] and the J2EE model uses native Java [13] defined interfaces and data structures. As a result, a business architect and/or developer must be familiar with a variety of access control technologies and platform languages in order to define the end-to-end access control in a heterogeneous environment.

In addition to the diverse vocabulary and specification languages utilized by the existing access control implementations, every layer of technology has an access control model. There are operating system models, database and messaging infrastructure models, and component technology models. The role-based access control (RBAC) has become popular as the access control model for component platforms. RBAC has enjoyed success because it is much more flexible and scalable than the user or group based models and is implemented in most of the available component platforms. It is not, however, sufficient to support many complex business scenarios. For example, the access control policies may require an assessment of additional environmental factors such as time, location, relationships or credit limits which may supplement an RBAC policy. For this reason, RBAC does not provide a complete unification model, rather a specific instance of a model.

There is also an issue related to expressing the access control rules. Unfortunately even when the access control is considered during analysis and design, the requirements are typically expressed in a natural language as business rules. That is, the focus is often to identify the access policy, not to architect the system so that it can accommodate dynamic policy changes. If (when) access policy changes, it becomes a part of the application project to modify the software to update the rules. This adds to the complexity of the access control architecture and makes it impossible to change the access policy without software changes. There are products which support model driven techniques, however, the access control mechanisms are typically expressed only in the platform specific manner (via application code, servlet filters, IIOP interceptors, J2EE deployment descriptors or product specific mechanisms such as graphical interfaces and proprietary API's). Thus, there is no standard way to define the access decision points and/or policy in a platform independent model such that it could be applied consistently across multiple technologies.

The end result is that the task of protecting business resources is increasingly being pushed to the business application developer. Of course, each level of access control still exists and must be administered. A single "application" typically has many "userids" (perhaps the same, perhaps different)" that are utilized in providing the access control across the application. As an example, the JAVA Blueprints Pet Store [25] has multiple userids that must be defined at different infrastructure levels before the application will execute successfully. This sample explores the "best practices" in a system integration architecture utilizing Web Services, Java Components (EJBs), Messaging (JAX/RPC, IIOP, and JMS) and Connectors (JDBC). In addition to multiple userids defined in deployment descriptors, the Pet Store application also supports self-registration of userids that are application specific and completely unknown to J2EE, the web server or the operating system. This "best practices" blueprint architecture documentation suggests that e-business applications must manage userids and the access control [26]. This is an example of the trend of pushing the access control into the application layer. That is, application architects and developers are being forced to include user management, access policies, and programmatic access control logic within their business software. This forces the expenditure of precious business developer resources on building application specific access control infrastructure for managing user information and access control policies, thereby, increasing the cost and time required to create the application.

A component infrastructure that requires exposing knowledge of the underlying access control model to the business developer (via programmatic API's such as isCallerInRole or isUserInRole) has made it difficult to hide the diversity inherent in the access control models when more complex access control policies are required. Although vendor products may extend the RBAC model and/or implement proprietary mechanisms to support more sophisticated access policy, in the absence of an AC-PIM as a reference model for access control, the task of understanding and comparing product features becomes difficult. The task of creating and maintaining consistent policies is also very difficult while a consolidated auditing is near impossible.

Access policies may often change and/or be governed by the legislation that differs from location to location. Business developers should not be required to understand those policies but unfortunately this is what happens during today's application development process. These issues limit the ability of a developer to use components in dynamic system compositions where the access control policy may be significantly different from what was provided in the original usage of that component. We will explore how the proposed model shifts the majority of this work to the provider of the infrastructure authorization service software and discuss how future component infrastructure could assist in assuming more of this responsibility.

4. Goals for Model Driven Access Control

To fully realize the potential of Model Driven Architecture for the access control, an access control platform independent model and the mechanism for the parameterization of domain platform independent models with access control points must be standardized within the OMG Model Driven Architecture roadmap. The goal of the access control platform independent model (AC-PIM) is to provide an abstract view of the access control that can be utilized at the modeling level for the parameterization of domain models. This will enable transformations to access control platform specific models (AC-PSM) that incorporate access control points.

The paper begins the analysis necessary for the unification of the access control models by identifying the vocabulary and abstractions that can be standardized for the purpose of model parameterization (thus enabling a transformation to existing access control models) and the common feature support to ease the secure interoperability. Thus, the goal of the proposed research is the creation of a unifying AC-PIM from which existing security models can be mapped and/or bridged. This will simplify the task of the middleware when cooperation is required to meet the underlying security constraints (such as the delegation of credentials and/or requesting access control decisions based on a local policy). For the UniFrame project, a goal is to identify the work necessary for enabling the generation of access control bridges for heterogeneous system compositions. That is, we wish to provide the foundation for new research projects in using the generative techniques for access control and secure interoperability.

An additional goal is to define a PIM that is simple enough for the business people to understand. This will enable meaningful communications between the business system architects and the security architects by providing appropriate abstractions and a vocabulary. Thus, a person should not need to be a security domain expert to understand the concepts of the Access Control Platform Independent Model. For this reason, if the security community uses multiple terms for similar concepts, the choice of this work is to use the one that is most likely to be familiar to a business person, or to introduce a new term that can be mapped to the more technical security term during transformation.

It is also a goal of this work to unify existing access control mechanisms while providing abstractions that enable future innovation. Hence, the proposed model should be flexible enough to support the authorization requirements of the future infrastructures such as the Grid. To do this, a Model Driven Access Control must support an architecture where access decisions are architecturally separate from business logic. Therefore, this paper challenges the current trend of pushing knowledge of the underlying access control model into application logic when programmatic access control is required.

5. Model Driven Access Control

A model transformation occurs when models are refined and details are added for the purpose of focusing on a particular implementation technology or an aspect of the domain model. Model transformations are used to document different "levels of abstractions", "viewpoints" or "aspects" of an information system. Models that comply with a specific meta-model may utilize generative techniques for the transformations; leveraging information that the generator knows regarding the target implementation platform and/or parameterizations provided by the software architect [4].

If Model Driven architecture is to reach it's full potential, the quality of service issues must have Platform Independent Models. For the security access control models, that means that an AC-PIM must be explored from which JAAS, RAD and SAML/XACML (and others) can be derived. This ensures that business people and software architects could utilize common vocabulary and syntax to define the access control architectures. This paper supports the development of an access control PIM expressed in UML that can be the catalyst for the unification of the existing access control PSMs.

The definition of this model is critical to support the development of MDA tools that generate security access control points as a part of the infrastructure that bridges technology platforms and hopefully will provide a catalyst for innovations in the access control standards that push much of the application level access control down into infrastructure containers and/or communications layers. Finally, the adoption of a common model will enable the migration of existing access control implementations to a more consistent access control infrastructure. Thus, it should be possible to standardize using the experience-based design patterns and the mappings that enable true Enterprise Security Services to be integrated at all levels of computing infrastructure (Security related design patterns for scalability which can be utilized as part of model transformation were discussed in earlier work [2]). Model Driven Security means allowing the security contracts to be modeled as a part of the component contract, thus enabling the security context to be managed end-to-end by the UniFrame infrastructure. This will enable manageable access control and auditing regardless of whether the system composition was statically or dynamically generated.

6. Proposed Platform Independent Model

There are five principles that this model provides in support of Model Driven Access Control:

1) Access Control Points should be identifiable via parameterization of the domain PIMs with a single model element.
2) Protected Resources should be identifiable via the parameterization of the domain PIMs with a single model element.
3) An access policy should be defined, developed and managed separately from the application business logic. Access policy rules and access policy models must be able to evolve without any modification of the business software.
4) The policy model (role-based, user-based, code-based...) should not be exposed to the business application developers. That is, the business logic should have absolutely no knowledge of the access control model utilized to make access decisions.
5) The access control platform independent model (AC-PIM) must provide abstractions that are (as much as possible) consistent with the existing commonly used Platform Specific Models for access control. If not, it will not be acceptable to the user or vendor community and the work to produce it will be purely academic with no long-term impact.

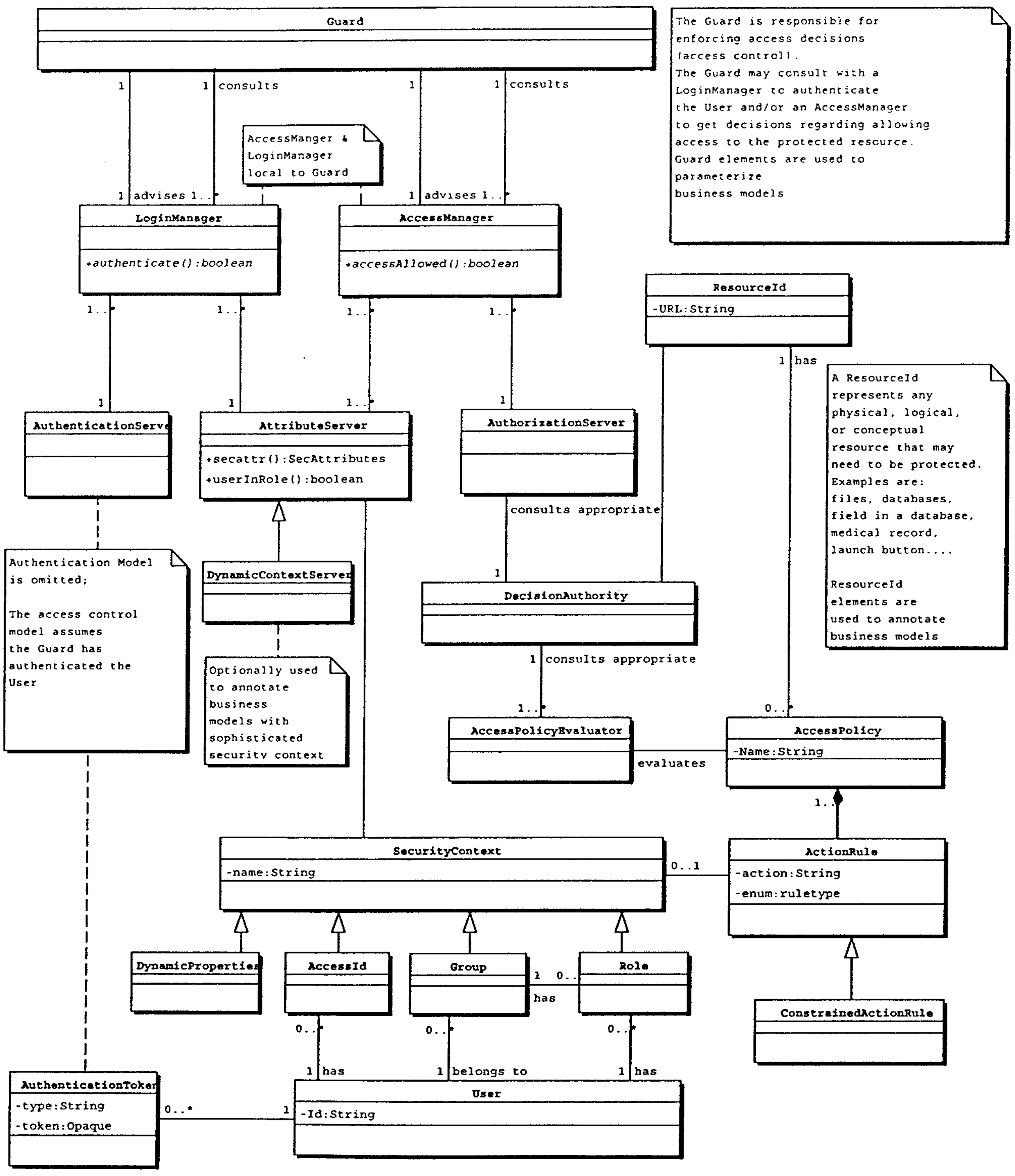

Guard
consults
advises
1..*
LoginManager
+authenticate():boolean
AccessManager
+accessAllowed():boolean
AccessManger & LoginManager local to Guard
The Guard is responsible for enforcing access decisions (access control). The Guard may consult with a LoginManager to authenticate the User and/or an AccessManager to get decisions regarding allowing access to the protected resource. Guard elements are used to parameterize business models
ResourceId
-URL:String
has
A ResourceId represents any physical, logical, or conceptual resource that may need to be protected. Examples are: files, databases, field in a database, medical record, launch button....
ResourceId elements are used to annotate business models
AuthenticationServer
AttributeServer
+secattr():SecAttributes
+userInRole():boolean
AuthorizationServer
consults appropriate
DynamicContextServer
DecisionAuthority
Authentication Model is omitted;
The access control model assumes the Guard has authenticated the User
Optionally used to annotate business models with sophisticated security context
AccessPolicyEvaluator
evaluates
AccessPolicy
-Name:String
SecurityContext
-name:String
0..1
ActionRule
-action:String
-enum:ruletype
DynamicProperties
AccessId
Group
Role
ConstrainedActionRule
belongs to
AuthenticationToken
-type:String
-token:Opaque
0..*
User
-Id:String

Figure 1 – Platform Independent Model for Access Control (AC-PIM)

To support these principles, we must explore the minimal knowledge that must be provided via parameterization during modeling. This includes:

1. Identification of Resources that might require protection (note – the architect may not know if a resource is protected and is only giving it an identifier so that the security policy can be defined at some future time).
2. Identification of the points within the application architecture where the access control checks should be made.
3. Identification of the application specific context /attribute information that might be needed at the point where an access control check is made (for example, withdrawal amount or credit limit or a relationship between a requestor and the information being requested).

The proposed AC-PIM model is shown in Figure 1. Although additional details related to the semantics and division of responsibility will need to be finalized in future work, this paper presents the initial design for the Access Control Platform Independent Model that will be used by the UniFrame team to progress unification of heterogeneous access control solutions. This model will evolve with that work. The modeling elements that represent the information that must be defined as parameterization of the domain model as discussed above are:

1. The PIM modeling element (object) that represents the resource and manages the identity is **ResourceId**. Note that there is not a model element for the protected resource as it is outside the scope of the AC-PIM. It is only present "by reference" in the model.
2. The PIM modeling element (object) that represents the access control points is a **Guard**.
3. The PIM modeling element (object) that supports a dynamic use of the context information in making the access decisions is a **DynamicContextServer**.

A ResourceId may represent any physical, logical or conceptual resource or a group of resources. For example, a ResourceId may represent a panel or button on a GUI, a feature of a cell phone, an operating system, an individual machine, an instance of a field in a file or database, an entire file or database, a C method invocation, an RMI or CORBA operation on an object, a process or application, or a concept such as "emergency room patient" or "psychiatric record". It is this "id" that represents the resource and is the target when requesting access decisions. The actual "resource" is stewarded by an application (or a real person in the case where it is a physical resource such as an x-ray film).

A Guard is inserted at every point within the application (or infrastructure) architecture where access control checks should be made and enforced. The decision regarding where to insert one or more Guard(s) will be made during the parameterization of the domain model prior to transformation to a platform specific technology for the components or system composition. Some technology platforms will provide Guards as part of the infrastructure (thus their meta-models already include one or more Guards). Others will require that Guards be manually added. Guards may be implemented by the operating system(s), messaging systems, infrastructure services, middleware, component containers, software components or applications. The Guard is responsible for ensuring that the user is authentic – that they are indeed who they claim to be (the authentication process is not covered by this paper – it is expected that any well accepted and popular methods of authentication would suffice) and that the user is authorized to access the protected resource. A Guard is typically a software piece protecting an electronic resource, but could also be a person who acts on the advice of software. The most important fact to understand is that it is the Guard and not the authorization service that is responsible for enforcement of the access control.

The ResourceId and Guard are the objects that will always be required to parameterize both domain models as developed by the business system architect, and the infrastructure component models used in the model transformation. The domain models must identify the resources that need to be protected and they must identify the points within the domain model where Guards should be activated. This parameterization will be utilized as a part of the model transformation to ensure that the access control checks are placed into the system at the appropriate points. In a similar way, infrastructure that enables dynamic system compositions (via the Web for example) must also support the dynamic identification of the protected resources and the insertion of Guards to protect them.

It is also anticipated that that some sophisticated business applications will require the ability to plug-in custom context servers that augment the infrastructure provided security services with an application specific security context. This plug-in ability is a requirement of an implementation of a compliant access control system. The DynamicContextServer provides the interface to support this feature. Architecturally this context server remains separate from the business logic and could be developed independently.

For the purposes of enabling Model Driven Access Control (via MDA tools), the remainder of the access control PIM shown in Figure 1 does not need to be exposed. We are, however, suggesting that the full model should be progressed as a mechanism for understanding and unifying the behavior of existing access control models and providing a consistent model that can be used as a basis of a reference model and vocabulary for future access control model evolution. For that reason, we define the full model in this paper. Now we will explore the architecture of the Access Control Platform Independent Model and how it supports the principles we outlined.

Consistent with most of the access control systems of today, a User may be either a person or a software component. The User is simply the requestor of access to a protected resource. When a user makes such a request, a Guard makes and enforces an access decision regarding whether to allow the User to access the protected resource that may be information or application feature(s). A ResourceId represents this protected resource.

The Guard has access to a local LoginManager and AccessManager for consultation purposes. The LoginManager and AccessManager are inserted into the model as locality constrained objects that provide an architectural support for location transparency and to address the need for high performance access control solutions; for example an implementation may support caching of the information stored in shared repositories and consolidation of multiple sources of the security information. These objects also support a unified application-programming interface (API) for requesting security authentication and authorization. This provides a plug-in point for vendors to integrate their solutions into heterogeneous environments.

The LoginManager's responsibility is to provide advice to the Guard on whether or not the requestor is who they claim to be. To do this, it uses an Authentication Service. Authentication Services may utilize diverse authentication technology including userid/password, X509 certificates, ticket-based (Kerberos), or token-based authentication. We will not explore a common authentication model in this paper except to note that this technology is significantly more mature in terms of allowing pluggable authenticators than authorization servers. The LoginManager has access to SecurityContextServer(s) for obtaining additional security context information if required during the process of authentication.

The AccessManager's primary responsibility is to provide advice to the Guard on whether or not the User should be should be allowed access to the resource(s) identified by the ResourceId. The AccessManager has access to one or more AttributeServer(s) and one or more AuthorizationService(s).

For a typical initial access request, the Guard will consult with the LoginManager to get advice on whether or not to trust the identity of the requestor and then consult with the AccessManager to determine if an access should be allowed to the requested resource(s). Subsequent requests for the resources from the same resource (typically within the scope of a session) would result in consultation only with the AccessManager. The Guard may also decide to trust an authentication token obtained by another Guard, which has been made available to it, and omit the consultation process with the local LoginManager. In this case, it would only consult with a local AccessManager. It is also possible that the Guard may determine that the resource is not protected at all and simply allows access to the resource without further consultation with either manager. Of course, a Guard may be disabled or removed which will allow access to all users, or may be inserted with a policy that causes it to deny access to all users.

It is important to understand that a Guard has the authority to accept or reject the advice of LoginManager and/or AccessManager(s) - although in practice this is typically not a good idea. This provides the flexibility to insert custom Guards in locations where normal application access control mechanisms must be disabled (such as emergency rooms where the information normally required to make decisions may not be readily available) or to temporarily deny access to everyone without any modification of the access policy. It is also important to note that the Guard does not have any knowledge of the access policy model. For example, if the underlying policy is role based access control (RBAC), the guard is not aware of the roles that may be required to access a resource. The Guard simply asks if an access is allowed or not by consulting with the AuthorizationService. A single implementation of a guard may therefore be used with multiple underlying access control models. This also supports our principle that the policy model (role-based, user-based, code-based, etc.) should not be exposed to the business application developers (as Guards will be provided by the application developers in environments where tools are not yet available to generate them). This is in contrast with current J2EE/EJB and Web Server programmatic security facilities which expose the role-based access control model by forcing a Guard to know the roles that are required and call "EJBContext::isUserInRole" or "HTTPServletRequest:: isCallerInRole" to make access decisions. The insertion of a local AccessManager into these architectures removes this requirement by allowing the Guard to call "AccessManager::access_allowed".

The AuthorizationServer, DecisionAuthority or AccessPolicyEvaluator(s) requires the security context information. The AccessManager must have this context information before consulting with an AuthorizationServer. Examples of the security context are security attributes such as groups, roles or access ids. Other examples of the security context are dynamic properties (such as the current balance on a checking account) that may be necessary to make an authorization decision. Such dynamic properties would be provided via a DynamicContextServer. By placing this in the access control architecture, a common design pattern is created that maintains the separation of the application logic and the access control logic and allows access policy to be evolved separate from the underlying business logic.

The AuthorizationServer consults an Access DecisionAuthority whose role is to combine the access decisions made by the PolicyEvaluators where multiple policies are in effect. For example, there may be a legal policy and an administrative policy that disagree. It would be the AccessDecisionAuthority that would be responsible for resolving any such conflicts. Simple AccessDecisonAuthority's would require consensus; more complex authorities might understand precedence rules. A PolicyEvaluator is responsible for evaluation of access policy. A PolicyEvaluator typically can evaluate any policy that follows a particular policy model (for example, role-based, access control lists or clearance based). Alternatively there may be PolicyEvaluators created for particular domains such as legal policy or administrative policy.

In the AC-PIM, the concept of AccessPolicy is abstract. An AccessPolicy consists of one or more Rules that are constructed using the SecurityContext. An AccessPolicy is associated (by name) with a ResourceId. The reason for requiring that AccessPolicy is associated "by name" to the ResourceId is to enable maximum scalability. By using this indirection, a policy can be managed independently and associated with many different resources. This is an expansion of the design pattern that enabled RBAC to scale (which creates "roles" that are assigned to Users and creating policy based on roles instead of individual users). This is more scalable than current deployment descriptors that require a redefinition of access policy rules in the deployment descriptor for each protected resource.

7. Transformation of the AC-PIM to Existing Platform Models

Figure 2 provides an overview of the models and aspects that must be considered as we look at the transformation of the AC-PSM to an AC-PSM for three existing Platform Specific Models.

1. The OASIS Access control model as defined in Security Assertion Markup Language (SAML) and eXtensible Access Control Markup Language (XACML).
2. The OMG access control model as utilized by the Resource Access Decision Facility (RAD).
3. The Java access control model as defined in the Java Authentication and Authorization Service (JAAS).

XACML is an XML specification for expressing policies. This specification defines an XML schema for an extensible access control policy language. As a result, it defines a standard vocabulary for the domain of access control policy. SAML is an XML-based security standard for a protocol to exchange the authentication and authorization information. This specification also defines the syntax and semantics for the XML-encoded SAML assertions about authentication, attributes and authorization.

	AC-PIM	JAVA / J2EE JAAS (JAAS PSM)	OMG RAD (RAD PSM)	OASIS SAML/ XACML (XACML PSM)
Access Control Model	Policy Based	Principal Based Access Control	Policy Based Access Control	Policy Based Access Control
Model Language	UML	Java Interfaces	ISO IDL	W3C XML
Description of Access Decision Model supported	An AccessManager uses a DecisionAuthority to make a decisions based on input from AccessPolicyEvaluators that may evaluate multiple policies. Named Policy is associated with ResourceIds	An AccessController determines if the Subject associated with the AccessControlContext has the required permission. Principals associated with the subject are matched against an application's required roles and permissions for the action are checked.	An AccessDecisionObject is passed the ResourceName, the requested operation on the resource, and SecurityAttributes. It locates the PolicyEvaluators and DecisionCombinator. One or more policies may be evaluated and combined by the combinator. Named Policy is associated with ResouceIds.	A Policy Decision Point (PDP) uses AuthorizationPolicy (gathered via PolicyRetrievalPoint) and evaluates it and makes an access decision which is provided via an AuthorizationAssertion
Policy Format standard	Not defined - Policy is named and associated with resources "by name". Policy format is encapsulated and is not standardized	XML - Policy is defined via grant statements in deployment descriptors that grant Permissions for actions to Security Principals (user/role)	Not defined - Policy is named and associated with resources "by name". Policy format is encapsulated and is not standardized	Policy is expressed via XACML statements as AuthorizationAssertions XACML provides a policy exchange language (in XML)
Access Decision API provided	The Guard calls AccessManager:: access_allowed()	The client sets AccessControlContext by invoking operation via Subject.doAs	The client (serving as a Guard) calls AccessDecision:: access_allowed	
Alternative native API	AttributeServer:: userInRole	EJBContext:: IsUserInRole		HttpRequest:: isCallerInRole
Infrastructure Guards (access control points)	Inserted via business model parameterization and generated by MDA tools	J2EE Web and EJB Containers and J2EE Connectors use JAAS Subject.doAs api	CORBA Security Service / CSIv2 may offer guard via interceptors.	Web Container and Servlet container is a SAML PolicyEnforcementPoint (PEP)
Application Guards (access control points)	Guard may be inserted by developers at identified access decision points	Developer may insert access decision point via Subject.doAs – java runtime then is used as guard	Developer may create a guard that uses the AccessDecisionObject:: access_allowed to get decisions.	Developer may insert SAML PEP in Valves or Filters or application code.
What the application components must identify	Guard insertion points, ResourceIds; actions on Resource; Optionally a custom SecurityContextService	References to external resources accessed References to inter-component calls made Ids of all role names if isUserInRole is used	ResourceNames ; actions on Resource; Optionally: custom DynamicAttributeService and/or custom PolicyEvaluator	References to external resources accessed References to inter-component calls made Ids of all role names used in isCallerInRole

Figure 2 – Contrasting the AC-PIM with existing Platform Specific Model

Table 1 indicates the conceptually similar modeling elements that need to be explored in an AC-PIM to XACML Platform Specific Model transformation:

AC-PIM	XACML-PSM
Guard	Policy Enforcement Point (PEP)
AccessManager	
LoginManager	
AuthenticationServer	AuthenticationAuthority
AttributeServer	AttributeAuthority
AuthorizationServer	AuthorizationAuthority
DynamicContextServer	PolicyInformationPoint
DecisionAuthority	Policy Decision Point (PDP)
ResourceId	URI reference

Table 1: Transformation to Key XACML Elements

Java and OMG CORBA share a security specification for secure interoperability. Common Secure Interoperability Specification, Version 2 (CSIv2) [27] supports the protocol necessary for infrastructure and applications to obtain the security context information necessary to leverage the Resource Access Decision Facility (RAD) for fine-grain access control. RAD provides a uniform service to assist in implementing infrastructure or application level access control where the protected resources may be physical, logical, or conceptual or understood only within the context of the business application. RAD was designed for use in multiple technology environments and addresses the problems of enterprises who have access control policy that is defined by privacy and confidentiality legislation (such as healthcare, telecommunications, and finance) These domains demand more sophisticated access control policies than what can be provided by infrastructure security.

Table 2 indicates the conceptually similar modeling elements that need to be explored in an AC-PIM to RAD Platform Specific Model transformation:

AC-PIM	RAD-PSM
Guard	RAD client
AccessManager	
LoginManager	
AuthenticationServer	AuthenticationService
AttributeServer	SecurityContext
AuthorizationServer	AccessDecisionObject
DynamicContextServer	DynamicAttributeService
DecisionAuthority	DecisionCombinator
ResourceId	ResourceName

Table 2: Transformation to Key RAD Elements

The Java™ Authentication and Authorization Service (JAAS) is a package that enables services to authenticate and enforce access controls upon users. JAAS authorization extends the existing Java security architecture that uses a security policy to specify what access rights are granted to executing code. JAAS authorization augments the existing code-centric access controls with new user-centric access controls. Permissions can be granted based not just on what code is running but also on who is running it. Permissions can be granted in the policy to specific Principals.

Table 3 indicates the conceptually similar modeling elements that need to be explored in an AC-PIM to JAAS Platform Specific Model transformation:

AC-PIM	JAAS-PSM
Guard	SecurityManager
AccessManager	Subject
LoginManager	LoginContext
AuthenticationServer	LoginModule
AttributeServer	Subject
AuthorizationServer	AccessController
DynamicContextServer	
DecisionAuthority	
ResourceId	Resource-ref

Table 3: Transformation to Key JAAS Elements

8. Conclusion

This paper proposes that access control patterns (in the form of a platform independent model) be utilized as a part of the component architectures to simplify the task of generating middleware that assumes the responsibility for the access control decisions that previously were tedious (or near impossible) to protect without the involvement of the application logic and the application developer. It proposes a Platform Independent Model that can be leveraged in a Model Driven Approach. While the full definition and standardization of such a security model is beyond the scope of this research project, this initial investigation indicates that the development of such a model is feasible. We are hopeful that this research will lead to the standardization of an Access Control Platform Independent Model (AC-PIM) under the OMG MDA process.

9. Future Work

Future work is planned to examine the problems associated with the protection of the fine-grain features and information resources of a typical Web-enabled business application. The case study, to be created for this purpose, will examine a Web-enabled business application architecture including a Web tier, an application logic tier, and an enterprise resource tier. Since an application may require that access to a resource in the enterprise resource tier be controlled based on the credentials of the Web user, it will explore the authorization and access control issues related to managing the security context across a multi-tier environment. A model-based solution will be proposed for the UniFrame project.

This case study will be used to validate the AC-PIM by completing the semantics of the models and the transformation to existing models. The proof of concept will take a well-known application (the Java Blueprints Pet Store), expressing the Pet Store domain model in UML and parameterizing it with the AC-PIM by identifying ResourceIds and inserting Guards that conform to the AC-PIM. A transformation of the model to the associated PSM that includes access control will then be progressed. An analysis of this manual transformation will serve as the foundation of code generators that mechanize the process. A final goal is to progress a standard Platform Independent Model for Access Control within the OMG community that can be leveraged by Model Driven Architecture tools.

We are currently involved in research in the use of formal methods for quality of service analysis in component-based distributed computing [28] and would like to investigate how formal methods might be leveraged in the access control domain. In addition, we hope to define future research projects that collaborate with groups doing natural language research and experiment with natural language processing of the access control requirements [29]. We also hope to collaborate with research groups that are using Aspect Oriented computing in Model Driven Architecture projects to determine if weaving techniques can be used to introduce access control logic during model transformation and at system composition time.

10. References

[1] Rajeev R. Raje, Barrett Bryant, Mikhail Auguston, Andrew Olson, Carol Burt. 2001. *A Unified Approach for the Integration of Distributed Heterogeneous Software Components*, Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration, pp: 109-119.

[2] Carol C. Burt, Barrett R. Bryant, Rajeev R. Raje, Andrew Olson. Mikhail Auguston. 2002. *Quality of Service (QoS) Standards for Model Driven Architecture.* Proceedings of the 2002 Southeastern Software Engineering Conference, pp. 521-529.

[3] Carol C. Burt, Barrett R. Bryant, Rajeev R. Raje, Andrew Olson, Mikhail Auguston. 2002. *Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models.* Proceedings of EDOC 2002, the 6th IEEE International Enterprise Distributed Object Computing Conference, pp 212-223.

[4] Object Management Group. *Model Driven Architecture Guide.* Technical Report. Document # omg/2003-05-01. Framingham, MA: Object Management Group. May 2003.

[5] ITU-T Recommendation X.812 (1995) | ISO/IEC 10181-3: 1995, *Information Technology -- Open Systems Interconnection -- Security Frameworks for Open Systems -- Access Control.*

[6]http:://www.globus.org.

[7] K. Keahey, V.Welch. 2002. *Fine-grain Authorization for Resource Management in the Grid Environment.* Proceedings of Grid2002 Workshop.

[8] http://www.oasis-open.org.

[9] OASIS. 2003. The XACML 1.0 Specification Set, available via http://www.oasis-open.org.

[10] OASIS. 2002. Security Assertion Markup Language version 1.0, available via http://www.oasis-open.org.

[11] http://www.omg.org.

[12] Object Management Group. 2001. *Resource Access Decision Facility.* formal/2001-04-01 (full specification) formal/2001-04-02 (OMG IDL). Available via http://www.omg.org/technology/documents/formal/ omg_security.htm.

[13] Sun Microsystems. 2002. *Java^{TM} Authentication and Authorization Service (JAAS) is part of Java 2 Platform Enterprise Edition Specification v1.4,* Available via ftp from www.java.sun.com. Sun Microsystems.

[14] http://www.jcp.org.

[15] Java Community Process. 2002. *JSR 115- Java^{TM} Authorization Contract for Containers.* Available for download from http://www.jcp.org.

[16] http://www.microsoft.com.

[17] http://www.gotdotnet.com/team/clr/about_security.aspx.

[18] D. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, R. Chandramouli, 2001. *A Proposed Standard for Role Based Access Control,* ACM Transactions on Information and System Security , vol. 4, no. 3.

[19] D.F. Ferraiolo and D.R. Kuhn 1992. *Role Based Access Control.* 15th National Computer Security Conference

[20] Numerous references are available at http://csrc.nist.gov/rbac/].

[21] M. Abrams, J. Heaney, O. King, L. J. LaPadula, M. Lazear, and I. Olson. 1991. *A Generalized Framework for **Access Control: Towards Prototyping the Orgcon Policy***, In Proceedings of National Computer Security Conference, pp. 257-266.

[22] Beznosov, Deng, Blakley, Burt, Barkley. 1999. *A Resource Access Decision Service for CORBA-based Distributed Systems.* ACSAC (Annual Computer Security Applications Conference).

[23] International Standards Organization (ISO). ISO-IEC 14772-2. IDL as standardized by the Object Management Group.

[24] World Wide Web Consortium (w3c). Extensible Markup Language (XML) is text format derived from SGML (ISO 8879). Available from www.w3c.org/XML.

[25] Sun Microsystems Blueprints program. Pet Store version 1.3.1_01 available for download from http://java.sun.com/blueprints.

[26] Sun Microsystems. *Designing Enterprise Applications with the J2EE Platform.* Chapter on Pet Store Security Architecture. available online from http://java.sun.com/blueprints

[27] Object Management Group. 2001. *Common Secure Interoperability version2 - Chapter 24 of CORBA/IIOP specification.* formal/2002-12-06. available via http://www.omg.org/technology/documents/formal/omg_security.htm.

[28] Chunmin Yang, Barrett R. Bryant, Carol C. Burt, Rajeev R. Raje, Andrew M. Olson, and Mikhail Auguston, 2003. *Formal Methods for Quality of Service Analysis in Component-Based Distributed Computing* to appear in Proceedings of IDPT 2003, the Seventh World Congress on IntegratedDesign and Process Technology.

[29] Chunmin Yang, Beum-Seuk Lee, Barrett R. Bryant, Carol C. Burt, Rajeev R. Raje, Andrew M. Olson, Mikhail Auguston. 2002. *Formal Specification of Non-Functional Aspects in Two-Level Grammar*, Proceedings of the UML 2002 Workshop on Component-Based Software Engineering and Modeling Non-Functional Aspects (SIVOES-MONA), http://www-verimag.imag.fr/SIVOES-MONA/uniframe.pdf.

From Natural Language Requirements to Executable Models of Software Components

Barrett R. Bryant, Beum-Seuk Lee, Fei Cao, Wei Zhao, Jeffrey G. Gray, Carol C. Burt
University of Alabama at Birmingham
{bryant, leebs, caof, zhaow, gray, cburt} @cis.uab.edu

Rajeev R. Raje, Andrew M. Olson
Indiana University-Purdue University-Indianapolis
{rraje, aolson}@cs.iupui.edu

Mikhail Auguston
Naval Postgraduate School
auguston@cs.nps.navy.mil

Abstract

The UniFrame approach to component-based software development assumes that concrete components are developed from a meta-model, called the Unified Meta-component Model, according to standardized domain models. Implicit in this development is the existence of a Platform Independent Model (PIM) that is transformed into a Platform Specific Model (PSM) under the principles of Model-Driven Architecture (MDA). This position paper advocates natural language as the starting point for developing the meta-model and representative domain models. The paper illustrates how natural language is mapped through the PIM to PSM using a formal system of rules expressed in a Two-Level Grammar (TLG). This allows software requirements to be progressed from domain logic to the implementation of components. The approach provides sufficient automation such that components may be modified at the model level, or even the natural language requirements level, as opposed to the code level.

1. Introduction

Model-Driven Architecture (MDA) [12] is an approach that separates the essence of an application from the specific middleware platform to which it is deployed. The basic approach is to define Platform Independent Models (PIMs) that express the application logic of components conforming to some domain (e.g., mission-computing avionics, safety-critical medical devices) and then to derive Platform Specific Models (PSMs) using a specific component technology (e.g. CORBA[1], J2EE[2], and .NET[3]).

Domain logic is typically expressed in natural language before a model is developed. Standardization of domains and their associated components is being undertaken by the Object Management Group (OMG)[4]. To facilitate the MDA approach to be used in practice, automated tools are needed to develop the domain-specifications from their requirements in natural language as well as to enable transformation from PIMs into PSMs. Furthermore, if MDA is to be used for constructing distributed real-time embedded (DRE) software systems, then the models must consider not only functional aspects of domain logic, but also non-functional properties, such as Quality-of-Service (QoS) requirements (e.g., latency and bandwidth requirements on a distributed video streaming system [23]). QoS attributes are not currently considered in the MDA framework.

UniFrame [31] is an approach for assembling heterogeneous distributed components, developed according to MDA principles, into a distributed software system with strict QoS requirements. Components are deployed on a network with an associated requirements specification, expressed as a Unified Meta-component Model (UMM) [30] in the Two-Level Grammar (TLG) specification language [4]. The UMM is integrated with generative domain models and generative rules for system assembly [10], which may be automatically translated into an implementation that realizes an integration of components via generation of glue and wrapper code. Furthermore, the glue/wrapper code is instrumented to enable validation of the QoS requirements [32].

This paper describes a unified method of expressing domain models in natural language, translating these into associated logic rules for that domain, application

[1] CORBA – Common Object Request Broker Architecture, http://www.corba.org

[2] J2EE – Java 2 Enterprise Edition, http://java.sun.com/j2ee

[3] http://www.microsoft.com/net

[4] http://www.omg.org

of the logic rules in building MDA PIMs, and maintaining these rules through development of PSMs. The complete mapping takes place using a formal system of rules expressed in TLG. This allows software requirements to be progressed from domain logic to implementation of components. It also provides sufficient automation such that components may be modified at the model level, or even the natural language requirements level, as opposed to the code level. Section 2 describes our previous work with TLG and its use as a specification language. The application of this to MDA is discussed in section 3. Finally, we conclude in section 4.

2. From Natural Language Requirements to Formal Models

To achieve the conversion from requirements documents to formal models requires several levels of conversion, as shown in Figure 1. First, the original requirements written in natural language are refined as a preprocessing of the actual conversion. This refinement task involves checking spellings, grammatical errors, consistent use of vocabularies, and organizing the sentences into the appropriate sections. The requirements are expected to be organized in a well-structured way, e.g. as laid out in [36] or as a collection of use-cases [16], and be part of an ontological domain [21]. Once they are structured in this way via human preprocessing, the remainder of the conversion occurs automatically. If modifications to requirements are needed, these modifications should be made to the requirements already preprocessed, not the original ones. Since we are allowing for specification of components that will be deployed in a distributed environment, Quality-of-Service attributes are also specified [38].

An example requirements specification from [19] is given below. This is a small piece of the Computer Assisted Resuscitation Algorithm (CARA) Infusion Pump Control System [37].

```
The host is powered up and all
software subsystems are available.
The pump software system is now in
the wait operating state. The patient
with IV/pump running is placed onto
the host. The pump cable is connected
to the host. The host now provides
power for the pump.
```

Next, the refined requirements document is automatically converted into XML[5] format. By using XML to specify the requirements, XML attributes (meta-data) can be added to the requirements to interpret the role of each group of the sentences during the conversion. The information of the domain-specific knowledge is specified in XML. The domain-specific knowledge describes the relationship between components and other constraints that are presumed to exist in requirements documents or too implicit to be extracted directly from the original documents [22]. The XML representation produced for the above specification is:

```
<class title = "Mode" meta = "mode">
  <class title = "wait state" meta
      = "submode">
    <paragraph meta = "pre_cond">
      <sentence>
        Host is powered up and all
        software subsystems are
        available
      </sentence>
    </paragraph>
    <paragraph meta = "pre_exec">
      <sentence>
        Patient with IV/pump
        running is placed onto the
        host
      </sentence>
      <sentence>
        Pump cable is connected to
        the host
    </sentence>
    </paragraph>
    <paragraph meta = "exec">
      <sentence>
        HOST now provides power for
        pump
      </sentence>
    </paragraph>
  </class>
  ...
</class>
```

A knowledge base is built from the requirements document in XML using natural language processing (NLP) to parse the documentation and to store the syntax, semantics, and pragmatics information. Each sentence is read by the system and each sentence is parsed into words. At the syntactical level, the part of speech (e.g. noun, verb, adjective) of each word is determined by bottom-up parsing, whereas the part of sentence (e.g. subject, object, complement) of each word is determined by top-down parsing [17]. The corpora of statistically ordered parts of speech (frequently used ones being listed first) of about 85,000

[5] XML – eXtensible Markup Language – http://www.w3c.org/xml

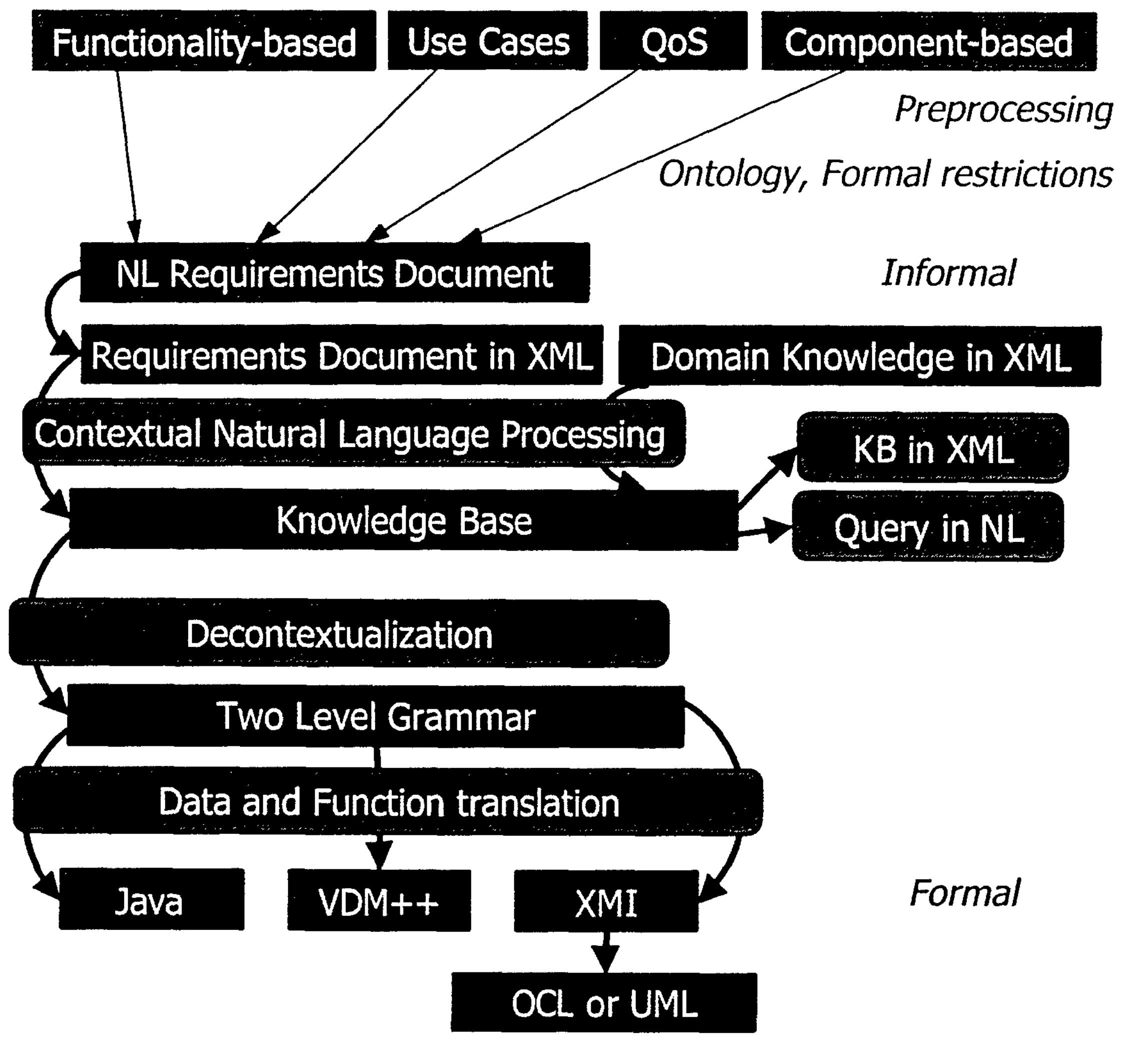


Figure 1. Natural Language Requirements Translation into Executable Models

words from [34] are used to resolve syntactic ambiguities in this phase. Also, elliptical compound phrases, comparative phrases, compound nouns, and relative phrases are handled in this phase as well. The knowledge base for the above example is shown in Figure 2.

Once the knowledge base is constructed, its content can be queried in NL. Next, the knowledge base is converted, with the domain-specific knowledge, into TLG by removing contextual dependencies in the knowledge base [20]. TLG is used as an intermediate representation to build a bridge between the informal knowledge base and the formal specification language representation. The name "two-level" in TLG comes from the fact that TLG consists of two context-free grammars interacting in a manner such that their combined computing power is equivalent to that of a Turing

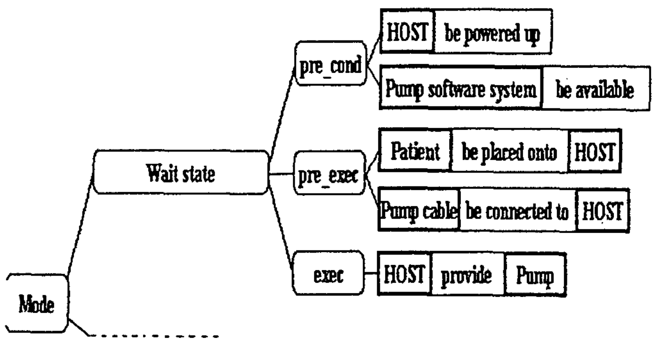


Figure 2: Knowledge Representation

machine. Our work has refined this notion into a set of domain definitions and the set of function definitions operating on those domains. In order to support object-orientation, TLG domain declarations and associated functions may be structured into a class hierarchy supporting multiple inheritance. The TLG specification produced for this example is:

```
class Mode.
  wait state :
    Host is powered up,
    Pump SoftwareSystem is
      available,
    Patient with IVPump running
      is placed onto Host,
    Pump Cable is connected to Host,
      Host provides Power for Pump.
    ...
end class Mode.
```

Host, Pump, SoftwareSystem (an attribute of Pump), Patient, IVPump (an attribute of Patient), Cable (an attribute of Pump), and Power have all been identified as objects in the analysis. In TLG, object and class names are denoted by being capitalized (and are in fact not distinguished, i.e., an object may be denoted using the corresponding class name, as an implicit declaration). Verbs and other words are included in TLG to make up functions, e.g. "is powered up," "is available," etc.

As a final step in this process, the TLG code is translated into VDM++, an object-oriented extension of the Vienna Development Method [11], by data and function mappings. VDM++ is chosen as the target specification language because VDM++ has many similarities in structure to TLG and also has a good collection of tools for analysis and code generation. Once the VDM++ representation of the specification is acquired, prototyping can be performed on the specification using the VDM++ interpreter to validate the generated formal specification against the original requirements. Also, the formal VDM++ representation can be converted into a high level language such as Java or C++, or into a Rational Rose model in UML[6] [29] using the VDM++ Toolkit [15]. The VDM++ specification created for the above TLG is:

```
class Mode

instance variables
  private host : Host
  private pump : Pump
  private patient : Patient
  private power : Power

operations
  ...
  public waitState : () => ()
  waitState () == (
    host . poweredUp ();
    pump . softwareSystem ()
      . available ();
    patient . ivPump ()
      . running ();
    patient . placedOnto (host);
    pump . cable ()
      . connectedTo (host);
    host . provides (power, pump);
  );
  ...

end class Mode
```

The VDM++ class uses one instance variable to represent each object in the TLG specification. This VDM++ specification may be converted into the UML model shown in Figure 3. Using the XMI[7] format, not only the class framework but also its detailed functionalities can be specified and translated into OCL (Object Constraint Language) [35].

3. Integration with Model-Driven Architecture

The method of translating requirements in natural language into UML models and/or executable code (as described in the previous section) may be used to translate domain logic into formal rules. Experts from various application domains may express their specification in natural language and then use UniFrame to translate this into TLG rules via natural language processing. These rules are encapsulated in a TLG class hierarchy defining a knowledge base with the domain ontology, domain feature models (specifying the commonality and variability among the product instances in that domain), feature configuration constraints, feature interdependencies, operational rules, and temporal

[6] UML – Unified Modeling Language, http://www.omg.org/uml

[7] XMI - XML Metadata Interchange, http://www.omg.org/technology/documents/formal/xmi.htm

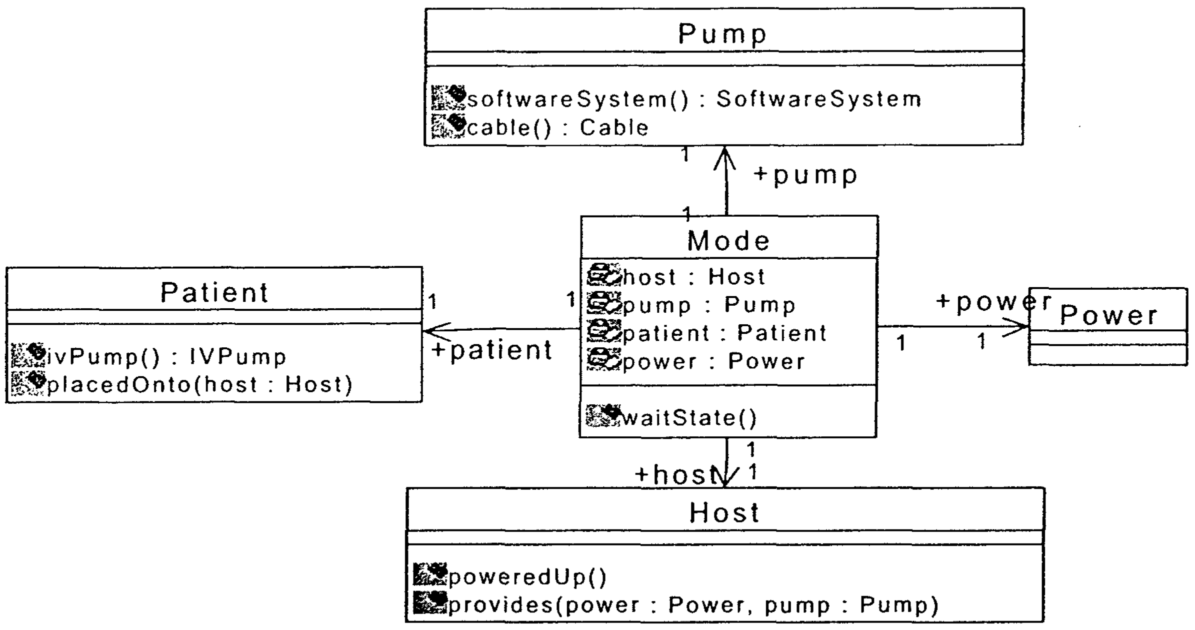


Figure 3: UML Representation of Requirements

concerns. TLG specifies the complete feature model including the structural syntax and various kinds of semantic concerns [39]. For example, assume that our application domain is for unmanned aerial vehicles (UAV's). The business domain will then include a feature model of a UAV, which includes specification of the various attributes and operations a UAV will have, such as responding to external commands and streaming video back to a satellite receiver [23]. In related work [8], we have investigated the construction of Generative Domain Models [10] using the Generic Modeling Environment [14]. This tool may also be extended with a natural language processor as a front end, i.e., by applying natural language processing to the domain model (represented in natural language), which can then extract feature model representation rules and then interpret those rules to generate a graphical feature diagram.

Platform Independent Models (PIM's) in MDA are based upon the domains and associated logic for the given application. TLG allows these relationships to be expressed via inheritance. If a software engineer wants to design a server component to be used in a distributed video streaming application, then he/she should write a natural language requirements specification in the form of a UMM (Unified Meta-component Model) describing the characteristics of that component. Our natural language requirements processing system will use the UMM and domain knowledge base to generate platform independent and platform specific UMM specifications expressed in TLG (which we will refer to as UMM-PI and UMM-PS, respectively). UMM-PI describes the bulk of the information needed to progress to component implementation. UMM-PS merely indicates the technology of choice (e.g. CORBA). These effectively customize the component model by inheriting from the TLG classes representing the domain with new functionality added as desired. In addition to new functionality, we also impose end-to-end Quality-of-Service expectations for our components (e.g., a specification of the minimum frame-rate in a distributed video streaming application). Both the added functionality and QoS requirements are expressed in TLG so there is a unified notation for expressing all the needed information about components. The translation tool described in the previous section may be used to translate UMM-PI into a PIM represented by a combination of UML and TLG. Note that TLG is needed as an augmentation of UML to define domain logic and other rules that may not be convenient to express in UML directly.

A Platform Specific Model (PSM) is an integration of the PIM with technology domain-specific operations (e.g. in CORBA, J2EE, or .NET). These technology domain classes also are expressed in TLG. Each domain contains rules that are specific to that technology, including how to construct glue/wrapper code for components implemented with that technology. Architectural considerations are also specified, such as how to distinguish client code from server code. We express PSMs in TLG as an inheritance from PIM TLG classes and technology domain TLG classes. This means that PSMs will then contain not only the application-domain-specific rules, but also the technology-domain-specific rules. The PSM will also maintain the QoS characteristics expressed at the PIM level (a related paper [6] explores the rules for this maintenance in more detail and [7] explores this issue for the QoS aspect of access control in particular). Because the model is expressed in TLG, it is executable in the sense that it may be translated into executable code in a high-level language (e.g. Java). Furthermore, it supports changes at the model level, or even requirements level if the model is not refined following its derivation from the requirements, because the code generation itself is automated.

Figure 4 shows the overall view of the model-driven development from natural language requirements into executable code for the previously described distributed video streaming application.

4. Related Work and Discussion

The idea of using natural language as the basis for developing software dates back at least 20 years. Abbott [1] pointed out that nouns correspond to the

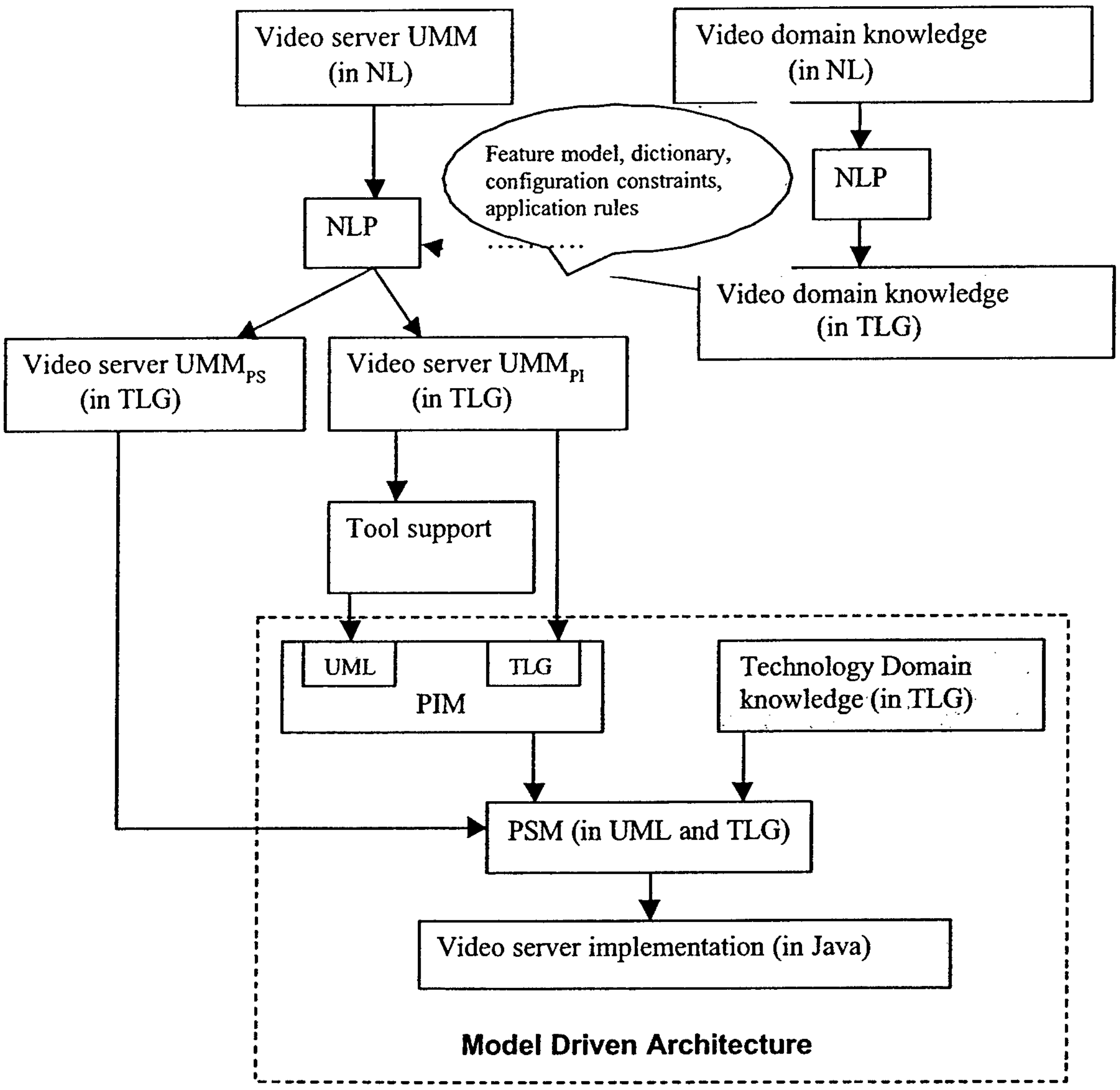

Video server UMM (in NL)
Video domain knowledge (in NL)
Feature model, dictionary, configuration constraints, application rules
NLP
NLP
Video domain knowledge (in TLG)
Video server UMM$_{PS}$ (in TLG)
Video server UMM$_{PI}$ (in TLG)
Tool support
UML
TLG
PIM
Technology Domain knowledge (in TLG)
PSM (in UML and TLG)
Video server implementation (in Java)
Model Driven Architecture

Figure 4. Integration of Two-Level Grammar with Model Driven Architecture

notion of a class in object-oriented programming terminology, direct references correspond to objects, while verb and attributes correspond to class operations, and the control flow within those operations is also often present in the action description. Rolland and Proix [33] developed an automated tool called OICSI[8], which facilitated the elicitation of requirements from natural language text and accompanying domain knowledge. Luisa Mich and her colleagues ([24], [25], [26]) have used a natural language processing system called NL-OOPS to analyze natural language requirements for the purpose of determining objects and their inter-relationships and construction of a corresponding object-oriented model. Nanduri and Rugaber [27] implemented a similar system for the purpose of validating an object-oriented model against the natural language requirements fom which it was derived. Ambriola and Gervasi [2] extended this idea to incorporate modeling and model checking to achieve a more formal validation (the authors use the term "semi-formal" to describe the validation approach, which eventually evolved into "lightweight formal methods" [13]). LIDA (Linguistic Assistant for Domain Analysis) [28] appears to be the most comprehensive system to date for assisting a software engineer to construct an object-oriented model from natural language descriptions, the emphasis being on domain models. Daniel Berry and his colleagues (e.g., see [3]) have also worked with the problem of analyzing natural language specifications and have identified a number of difficult problems in correctly implementing requirements based upon natural language.

Our work has focused on conversion of natural language to formal specifications in VDM++, which in turn may be converted into UML models or executable code. This paper has described an approach for unifying the ideas of expressing requirements in natural language, constructing Platform Independent Models for software components, and implementing the components via Platform Specific Models. The approach is specifically targeted at the construction of heterogeneous distributed software systems where interoperability is critical. This interoperability is achieved by the formalization of technology domains with rules describing how those technologies may be integrated together via the generation of glue and wrapper code. The processing of software requirements, construction of PIMs and PSMs, and specification of technology domain rules are all expressed in TLG, thereby achieving a unification of natural language requirements with MDA.

For future work, we will investigate aspect-oriented technology [18] as a mechanism for specifying crosscutting relationships across components and hence improving reusability of components and reasoning about a collection of components. Such aspects of components as functional pre/post conditions and QoS properties crosscut component modules and specification of these aspects spread across component modules. Preliminary work in defining an aspect-oriented specification language is very promising [9].

[8] French acronym for intelligent tool for information system design," also called ALECSI [33]

5. Acknowledgements

This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350, and by the U. S. Office of Naval Research under award number N00014-01-1-0746.

6. References

[1] Abbott, R. J., "Program Design by Informal English Descriptions," *Commun. ACM 26*, 11 (Nov. 1983), 882-894.

[2] Ambriola, V. and Gervasi, V., "Processing Natural Language Requirements," *Proc. ASE '97, 12th Int. Conf. Automated Software Engineering*, 1997, pp. 36-45.

[3] Berry, D. M. and Kamsties, E., "Ambiguity in Requirements Specification," *Perspectives on Software Requirements*, eds. J. C. Sampaio do Prado Leite and J. H. Doorn, Kluwer Academic, 2003, pp. 191-194.

[4] Bryant, B. R. and Lee, B.-S., "Two-Level Grammar as an Object-Oriented Requirements Specification Language," *Proc. HICSS-35, 35th Hawaii Int. Conf. System Sciences*, 2002, http://www.hicss.hawaii.edu/ HICSS_35/HICSSpapers/ PDFdocuments/STDSL01.pdf.

[5] Bryant, B. R., Auguston, M., Raje, R. R., Burt, C. C, and Olson, A. M., "Formal Specification of Generative Component Assembly Using Two-Level Grammar," *Proc. SEKE 2002, 14th Int. Conf. Software Engineering Knowledge Engineering*, 2002, pp. 209-212.

[6] Burt, C. C., Bryant, B. R., Raje, R. R., Olson, A. M., Auguston, M., 'Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models," *Proc. EDOC 2002, 6th IEEE Int. Enterprise Distributed Object Computing Conf.*, 2002, pp. 212-223.

[7] Burt, C. C., Bryant, B. R., Raje, R. R., Olson, A. M., Auguston, M., "Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control," *Proc. EDOC 2003, 7th IEEE Int. Enterprise Distributed Object Computing Conf.*, 2003, pp. 159-171.

[8] Cao, F., Bryant, B. R., Burt, C. C., Huang, Z., Raje, R. R., Olson, A. M., Auguston, M., "Automating Feature-Oriented Domain Analysis," *Proc. SERP 2003, 2003 Int. Conf. Software Engineering Research and Practice*, 2003 , pp. 944-949.

[9] Cao, F., Bryant, B. R., Raje, R. R., Auguston, M., Olson, A. M., Burt, C. C., "Assembling Components with Aspect-Oriented Modeling/Specification," *Proc. WiSME 2003, UML 2003 Workshop Software Model Engineering*, 2003, http://www.metamodel.com/wisme-2003/12.pdf.

[10] Czarnecki, K., Eisenecker, U. W., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[11] Dürr, E. H., van Katwijk, J., "VDM++ - A Formal Specification Language for Object-Oriented Designs," *Proc. TOOLS USA '92, 1992 Technology of Object-Oriented Languages and Systems USA Conf.*, 1992, pp. 263-278.

[12] Frankel, D.S., *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley Publishing, Inc., 2003.

[13] Gervasi, V. and Nuseibeh, B., "Lightweight Validation of Natural Language Requirements," *Softw. Pract. Exper. 32* (2002), 113-133.

[14] *GME 2000 User's Manual, Version 2.0.* ISIS, Vanderbilt University, 2001, http://www.isis.vanderbilt.edu/publications/archive/Ledeczi_A_12_18_2001_GME_2000_U.pdf.

[15] IFAD, The VDM++ Toolbox User Manual, 2000, http://www.ifad.dk.

[16] Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.

[17] Jurafsky, D., Martin, J., *Speech and Language Processing*, Prentice-Hall, 2000.

[18] Kiczales, G., et al., "Aspect-Oriented Programming," *Proc. ECOOP '97, 1997 European Conf. Object-Oriented Programming*, 1997, pp. 220-242.

[19] Lee, B.-S. and Bryant, B. R., "Automation of Software System Development Using Natural Language Processing and Two-Level Grammar," *Proc. 2002 Monterey Workshop Radical Innovations of Software and Systems Engineering in the Future*, 2002, pp. 244-257.

[20] Lee, B.-S. and Bryant, B. R., "Contextual Knowledge Representation for Requirements Documents in Natural Language," *Proc. FLAIRS 2002, 15th Int. Florida AI Research Symp.*, 2002, pp. 370-374.

[21] Lee, B.-S. and Bryant, B. R., "Contextual Processing and DAML for Understanding Software Requirements Specifications," *Proc. COLING 2002, 19th Int. Conf. Computational Linguistics*, 2002, pp. 516-522.

[22] Lee, B.-S. and Bryant, B. R., "Applying XML Technology for Implementation of Natural Language Specifications," *Comput. Syst., Sci. & Eng. 5* (September 2003), 3-24.

[23] Loyall, J., Schantz, R., Atighetchi, M., and Pal, P., "Packaging Quality of Service Control Behaviors for Reuse," *Proc. ISORC 2002, 5th IEEE Int. Symp.Object-Oriented Real-time Distributed Computing*, 2002, pp. 375-385.

[24] Mich, L., "NL-OOPS: From Natural Language to Object-Oriented Requirements using the Natural Language Processing System LOLITA," *J. Nat. Lang. Eng. 2*, 2 (1996), 161-187.

[25] Mich, L. and Garigliano, R., "The NL-OOPS Project: OO Modeling using the NLPS LOLITA," *Proc. NLDB '99, 4th Int. Conf. Applications of Natural Language to Information Systems*, 1999, pp. 215-218.

[26] Mich, L., Mylopoulos, J., and Zeni, N., "Improving the Quality of Conceptual Models with NLP Tools: An Experiment," Technical Report, Department of Information and Communication Technologies, University of Trento, Italy, 2002, http://eprints.biblio.unitn.it/archive/00000127/01/ 47.pdf.

[27] Nanduri, S. and Rugaber, S., "Requirements Validation via Automated Natural Language Parsing," *J. Manage. Inf. Syst. 12*, 2 (1996), 9-19.

[28] Overmyer, S. P., Lavoie, B., and Rambow, O., "Conceptual Modeling through Linguistic Analysis using LIDA," *Proc. ICSE 2001, 23rd Int. Conf. Software Engineering*, 2001, pp. 401-410.

[29] Quatrani, T., *Visual Modeling with Rational Rose 2000 and UML*, Addison-Wesley, Reading, MA, 2000.

[30] Raje, R. R., "UMM: Unified Meta-object Model for Open Distributed Systems," *Proc. ICA3PP, 4th IEEE Int. Conf. Algorithms and Architecture for Parallel Processing*, 2000, pp. 454-465.

[31] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., and Burt, C. C., "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," *Proc. 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration*, 2001, pp. 109-119.

[32] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., Burt, C. C., "A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components," *Concurrency Comput.: Pract. Exp. 14*, 12 (2002), 1009-1034.

[33] Rolland, C. and Proix, C., "A Natural Language Approach for Requirements Engineering," *Proc CAiSE '92, 4th Int. Conf. Advanced Information Systems*, 1992.

[34] Ward, G., "Moby Part-of-Speech II (data file)," 1994, http://www.gutenberg.net/extext02/mposp10.zip.

[35] Warmer, J., Kleppe, A., *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

[36] Wilson, W. M., "Writing Effective Natural Language Requirements Specifications," Naval Research Laboratory, 1999.

[37] Walter Reed Army Institute for Research (WRAIR), "CARA Specification: Proprietary Document," WRAIR, Dept. of Resuscitative Medicine, 2001.

[38] Yang, C., Lee, B.-S., Bryant, B. R., Burt, C. C., Raje, R. R., Olson, A. M., Auguston, M., "Formal Specification of Non-Functional Aspects in Two-Level Grammar," *Proc. UML 2002 Workshop Component-Based Software Engineering and Modeling Non-Functional Aspects (SIVOES-MONA)*, 2002, http://www-verimag.imag.fr/SIVOES-MONA/uniframe.pdf.

[39] Zhao, W., Bryant, B. R., Burt, C. C., Gray, J. G., Raje, R. R., Olson, A. M., Auguston, M. "A Generative and Model Driven Framework for Automated Software Product Generation," *Proc. CBSE 6, 6th Workshop Component-Based Software Engineering*, 2003, http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6/Proceedings/papersfinal/p31.pdf.

Assembling Components with Aspect-Oriented Modeling/Specification*

Fei Cao[1], Barrett R. Bryant[1], Rajeev R. Raje[2], Mikhail Auguston[3], Andrew M. Olson[2], Carol C. Burt[1]

[1]Department of Computer and Information Sciences
University of Alabama at Birmingham
{caof, bryant, cburt}@cis.uab.edu

[2]Department of Computer and Information Science
Indiana University Purdue University at Indianapolis
{rraje, aolson}@cs.iupui.edu

[3]Computer Science Department
Naval Postgraduate School
auguston@cs.nps.navy.mil

Abstract:
Component-Based Software Development (CBSD) offers a cost-effective means of software production with reduced time-to-market. Integration of heterogeneous components poses a non-trivial challenge in realizing this vision, which is further complicated in a distributed environment as a result of blurred functional and non-functional aspect[1] representation and management. We propose a two-level approach, i.e., to apply aspect-oriented component modeling/specification to handle the problem.

Keywords:
Aspect Orientation, Component Modeling/Specification, UniFrame, Weaving

1. Introduction

1.1 Background

Recent development in software component technology enables the production of complex software systems by assembling off-the-shelf components. This not only boosts productivity attributed to the reusability of components, but also improves cost-control and maintenance of software systems. Meanwhile, another hallmark of current software components is the heterogeneity in environment, language and application over distributed systems.

UniFrame [Raje01] is a framework for seamless interoperation of heterogeneous distributed software components. It is based on the Unified Meta-component Model (UMM) [Raje00] for describing components. A Generative Domain Model (GDM) [Czar00] is used to describe the properties of domain specific components and to elicit the rules for component assembly. Systems constructed by component composition should meet both functional and non-functional requirements such as the Quality of Service (QoS) [Brah02]. Towards the realization of the vision of the UniFrame project, an appropriate means for component modeling/specification is needed, which should be capable of:

* This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

[1] In this paper, "non-functional aspect", "non-functional-property" and "Quality of Service (QoS)" may be used interchangeably.

- representing the functional properties (including not only syntactic structure but also semantic behaviors) and requirements (pre/post condition, dependency, temporal constraints, etc.).
- representing the non-functional properties and requirements [Brah02].
- specifying the heterogeneity in terms of representing domain knowledge, e.g., technology domain, business domain, etc.

1.2 Current Issues

Assembly of heterogeneous distributed components will require glue/wrapper code to fuse them together. General practice leverages vendor-specific bridging products or applies hard coding, and both the functional and non-functional aspects of the assembled system tend to be blurred by this ad hoc treatment. We have applied Two-Level Grammar (TLG) as a formalism to specify various aspects of components [Brya02] based on UMM. Meanwhile, it has been brought to our attention such aspects of components as functional pre/post conditions and non-functional properties crosscut component modules and handling of these aspects spread across component modules. This poses some problems:

- reduced reusability of components. Component behavior may change in different contexts. The inter-relationship between components may also change under different business rules. The "Hard-coded" modeling/specification will be inadequate to capture the dynamics of components and component representations may have to be revised upon different environments
- blurred representation and management of functional and non-functional aspects of components. As those aspects are entangled with other aspects of components, reasoning for the integrated system based on those aspects will be hard to be carried out.

Aspect Orientation [Kicz97] provides a means to capture crosscutting aspects in a modular way with new language constructs. This makes us believe that augmenting our existent specification approach with aspect orientation can separate those crosscutting aspects intervening components, loosen the coupling between components, which will contribute to not only the reusability and evolution of component without changing the component itself, but also the manageability of component assembly. On the other hand, by using weaving technology, dynamic concerns can be "glued" into the composition of components. This paper will investigate the application of aspect orientation in the modeling/specification of components, in particular, the handling of their exported service and QoS of heterogeneous distributed components in the context of the UniFrame project.

This paper is organized as follows: Section 2 first gives an analysis of component assembly models. Section 3 presents our two-level, model-based, aspect-oriented approach for heterogeneous distributed component representation. Section 4 draws the conclusion.

2 Component Assembly Model Analysis

In [Shaw97], *component* and *connector* are proposed as building blocks of software architecture. The examples of component include clients, servers, databases; the examples of connector include procedure call, event broadcast, database protocols. The various kinds of combination patterns of component and connector form the collection of architecture styles.

From the perspective of component assembly, we use the connector concept as an abstraction for glue/wrapper codes necessary for component assembly, and analyze how the use of this abstraction makes the assembly process scalable. The approach of removing assembly logic from the component into the connector can increase the reusability of the component, reduce the complexity and boost maintainability. Meanwhile, assembly model analysis will contribute to the automation of this process. Based on the hierarchical relationship between component and connector in the assembly process, the assembly models can be categorized as follows:

1) the connector and component reside at the same level (Figure 1).
   This is the most common and simple assembly model, and conforms to most architecture styles listed in [Shaw97], such as *pipes and filter,* and *event system.* The connector here may be remote

method call, or event/message based communication for client/server architecture. This model is mostly seen in distributed component assembly.

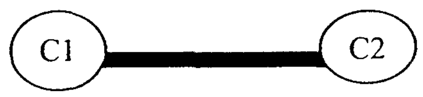


Figure 1: Component & Connector: Same Level

2) the components are contained in the connector. Figure 2 provides a COM[2] model.

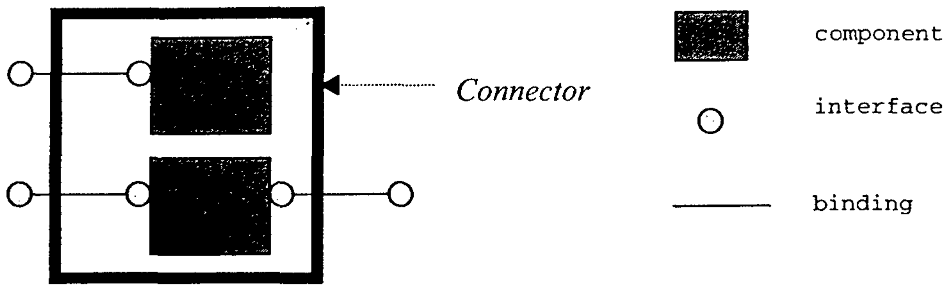


Figure 2: Connector as a Container

The connector acts as an infrastructure in the form of framework, which assembles components via *inversion of control*, such as in EJB[3], CCM[4]; or a package, using such way as manifest file to package components, such as in JavaBeans[5]. Also such connector in some cases plays the role as a container providing extra services for the components to leverage, such as security, transaction, life cycle management, persistence.

3) mixed form of the above two cases.
In this case, component assembly is comprised of a hierarchical process, the father assembly is derived from the assembly of the output of each child assembly process, in the form as described in either (1) or (2). Each child assembly process further is derived from their own child assembly process in either (1) or (2).

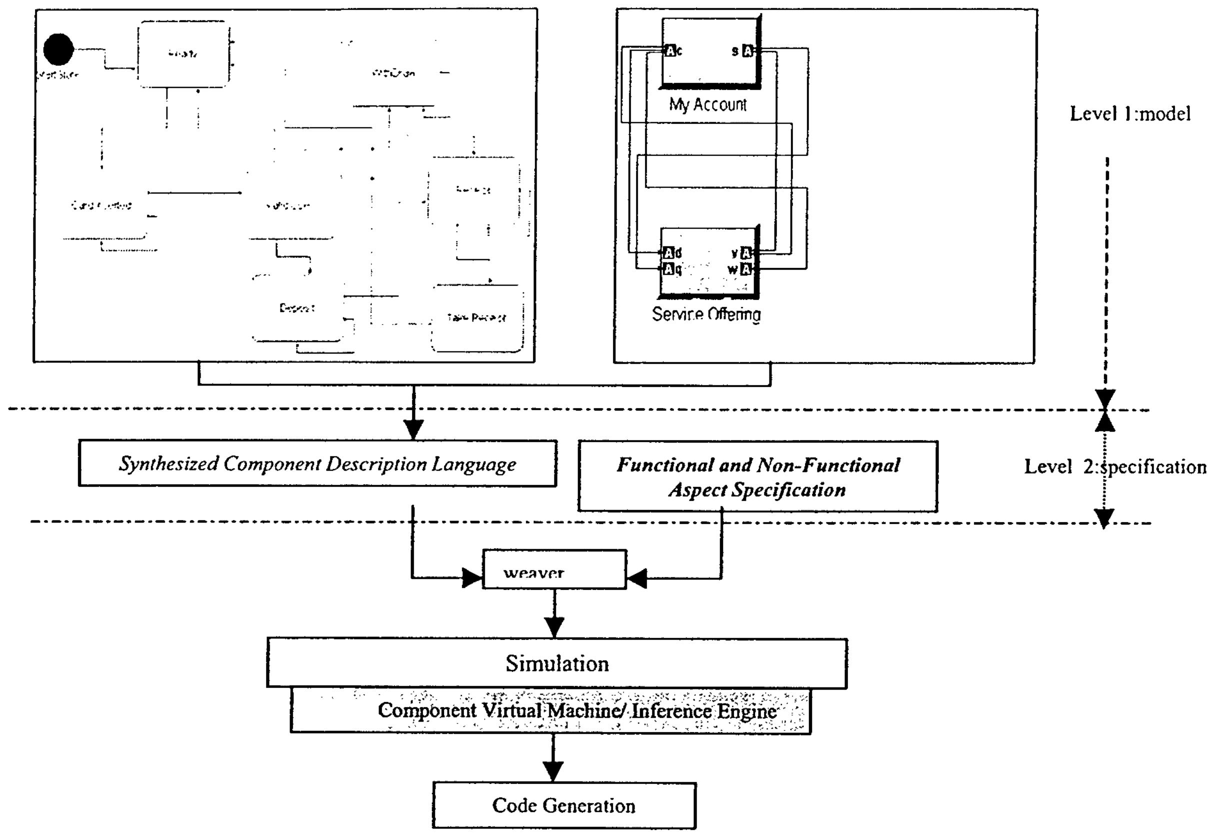
3. Two-level Component Modeling/Specification with Aspect Orientation

In light of prior assembly analysis, we propose a *two-level* approach toward an effort of component assembly by handling the modeling of the component and the specification of their interaction (aka. connector) separately: the first level is the modeling of heterogeneous components (their functional as well as non-functional properties [Brah02]) in graphical forms using some advanced CASE tools such as the Generic Modeling Environment (GME) [GME01]; the specification of inter-relationships between components and manipulations of the component model are included in the second level, which constitutes the connector module. The assembly of components for the production of the final system will be in an automatic fashion using an aspect weaver based on the modeling and specification. Figure 3 illustrates the process.

[2] COM: Component Object Model, *http://www.microsoft.com/com.*
[3] EJB: Enterprise Java Beans, *http://java.sun.com/products/ejb*
[4] CCM: CORBA® Component Model, *http://www.omg.org/cgi-bin/doc?orbos/99-07-01*
[5] *http://java.sun.com/beans/*

3.1 Level 1: Component Modeling

One of the Object Management Group (OMG) [6] initiatives is Model Driven Architecture (MDA®) [OMG01], i.e., by reverse engineering legacy systems and Commercial-Off-The-Shelf (COTS) components, software can be transformed into Platform Independent Models (PIMs). PIMs, in turn, will be mapped to Platform Specific Models (PSMs), such as CORBA[7], EJB, SOAP[8] and .NET[9]. In this way, legacy systems



My Account
Service Offering
Level 1:model
Synthesized Component Description Language
Functional and Non-Functional Aspect Specification
Level 2:specification
weaver
Simulation
Component Virtual Machine/ Inference Engine
Code Generation

Figure 3: Process of Aspect-Oriented Component Modeling/Specification

and COTS can be reintegrated into new platforms efficiently and cost-effectively. We embrace the same vision here by representing the software components with a model-based approach. However, such PIM model envisioned here is derived by creating meta-models specific to component modeling. In other words, we need to formulate the building blocks for describing component models. This includes the meta-model for business and technology domains [Zhao03]. But these are out of our scope here, which are actually the concerns of some organization such as OMG. Additionally, there should be a means in the component modeling level to represent the *join point* [Kicz97] in a component, which denotes the points that are affected by a particular crosscutting concern. In an AOP language such as AspectJ [Kicz01], *join points* are represented by referring to the syntactical constructs of the base program source. [Stei02] explores the representation of *join points* in UML models by marking affected model elements using UML tags. Here we may denote the *join points* by referring to the meta-information of model constructs. In that sense the

[6] *http://www.omg.org*
[7] *http://www.CORBA.org*
[8] SOAP: Simple Object Access Protocol, *http://www.w3.org/TR/SOAP*
[9] *http://www.microsoft.com/net*

join points here also represent domain knowledge and can serve as query parameters in search of specific components.

As is illustrated in the diagram, the first-level model will be transformed into the second level using a model-based approach consistent with the vision of MDA. This can be achieved easily using the meta-model information of the component models. In GME [GME01], this is realized by using the Builder Object Network (BON) framework for building interpreters, which traverses objects in the model tree by calling methods within the BON API and generates the Component Description Language (CDL), which also includes associated meta-model information to be used as the anchor of the join point.

3.2 Level 2: Component Specification

This level involves the creation of an Aspect Specification Language (ASL[10]) for describing crosscutting concerns in a separate way. Also a weaver is built to weave the ASL with CDL to generate targeted executable specification of components.

3.2.1 Constructs of ASL

In AspectJ [Kicz01], the aspect specification includes three elements: *pointcuts* to pinpoint the affected location of applications; *advice* to describe the actions that are applied to the *pointcuts*; the condition which governs how/when to apply *advice* to *pointcuts* using "before", "after", etc. To generalize for ASL, we need a means to specify:

1) join points.
2) behavior specification describing the actions to be performed.
3) policy on how the behavior is applied to join points.

(1) is as mentioned in 3.1, and is supposed to be specified in CDL. (2) and (3) will be provided in ASL.

3.2.2 Concerns Involved

This part will eventually evolve into a catalog of concerns to be handled in heterogeneous distributed component specification. For now the most distinct concerns involved will be:

1) gluing/wrapping of components.
   The gluing/wrapping of components is generally influenced by such aspects as platform and distribution. The component assembly process will be subject to evolution if components are deployed on a different platform/location. This dynamism can be well embraced by policy description in ASL. The pre/post condition as well as other constraint checking necessitated for the components to perform interaction (here, assembly) can be represented in the behavior specification under the corresponding policy. Obviously here the join points are contained in the involved components to be assembled.
2) QoS measurement.
   We also embed the non-functional aspects such as QoS measurement at the higher level specification of ASL, which will contribute to the measurement of QoS of the generated system at run-time. This is especially desired in a dynamic distributed environment, where a large amount of existent components may be exported for use, overall system QoS serving as the criteria to the filtering of service offerings among peer components. In [Augu95], event grammar is proposed to perform the system testing. We believe the introduction of the aspect-oriented approach will provide support to this effort, i.e., we can treat the QoS probing code as a behavior specification; the policy will govern how the probing code will be called at join points for dynamic measuring of QoS. The probing code will not be manually embedded in the points of interest, but rather using the weaver for dynamic instrumentation.

3.2.3 Simple Assembly Example using Aspect Orientation

To help clarify the aforementioned concepts, we give a simple example demonstrating how aspect orientation can be applied to component assembly. The ideas are adapted from aspectual components [Lieb99], in which aspects are decoupled from the base program by being defined as a generic aspectual

[10] Note this is nothing to do with the *Action Semantics Language* of OMG.

component, which is instantiated later over a concrete data-model. In this way, an aspect definition can be reused. Here we define aspectual component by capturing join points at the meta-model level of components.

Assume the component A is a banking domain client component hosted on Java RMI requesting some banking service from some server side. Below is the partial specification of its CDL:

```
A.0   Component A
A.1   Bankoperation:: Service.
A.2   Bank::BusinessDomain.
A.3   Platform::TechDomain.
A.4   Platform= "RMI".
A.5   Requires Bankoperations .
A.6   end Component A.
```

Note that right hand side of "::" denotes the *meta-type* of the left hand side. Line A.4 and A.5 are *hyper-rules*. *Meta-type* and *hyper-rule* are Two-Level Grammar notations. For more details of TLG, see [Brya02].

The above specification will be translated into a corresponding aspectual component:

```
B.0   aspect A
B.1   Bankoperation:: Service.
B.2   Bank::BusinessDomain.
B.3   expect Bankoperations.
B.4   expect wrap Argument. //usage interface
B.5   replace  Bankoperation:   //modification interface
B.6         if expected().getComponent().getPlatform()== "CORBA"
B.7         then  return expected().wrap("RMI").
B.8   end aspect A
```

Note those lines prefixed by **expect** denote operation signatures that are expected to be supplied with *advice*. In that sense the operation signatures here correspond to the join points in AOP. In the proposed approach here we only use meta-level types for the operation signature definition. Also the above **expected** keyword denotes something to be bound to join points. In line B.3, *Bankoperation* itself is meta-type in the banking business domain. Expected operations are either used (usage interface) or modified (modification interface, preceded with **replace**) in the aspectual component definition. For details please see [Lieb99]. Also lines B.6-B.7 provide *advice* (reimplementation) for the associated operations to be specified in the *connector* part below.

Assume the component B is a banking domain server component implemented in CORBA providing some banking services.

```
C.0   Component B.
C.1   Withdraw, Deposit:: Service;Port.
C.2   Bank::Domain.
C.3   Platform::TechDomain .
C.4   Platform= "CORBA".
C.5   end Component B.
```

Note in line C.1, the two types denoted in the right hand side of "::" means both withdraw and deposit are not *Services*, but also *Ports*, which means they are component services offered to external components.

The following is an ASL specification for component assembly.

```
D.0   connector A-B
D.1   Bankoperation=Withdraw, Deposit.     //join points
D.2   wrap(Argument): if (Argument.getname=="RMI")
D.3                {
D.4                       //provide wrapping specification for
D.5                              //RMI-CORBA inter-operation
D.6                   }
D.7    end  connector A-B
```

Note that lines D.2-D.6 further implement the *advice* part for the join points (here, *Withdraw and Deposit* operation). The body of *wrap* is ignored without loss of generality.

From the example illustrated in this section, we can see the interactions of two components can be separated by being handled in a module (here in the aspectual component definition, i.e. the "aspect A" module). Consequently the assembly process can be implemented by using a weaver to weave *advice* together with component specifications. As we can see in the body of "aspect A", it is straightforward for us to apply other concerns in between, e.g., we can call expected().precondition() wherever applicable in the **replace** function body to enforce some preconditions.

3.3 System-Level Simulation

We are investigating such program transformation tool as DMS[11] for building a weaver to weave CDL and ASL together, the output of which will be fed into the simulation phase to validate the functional system behavior against requirements before implementation code is generated and deployed. This simulation may be carried out by building a component virtual machine [Ducl02], which serves as an interpreter to interpret the weaved specifications; or by building rule sets based on requirement and then use some inference engine to validate the functional requirements. In this way, the assembled system will be functionally sound at an early phase. On the other hand, the generated applications, as they are probed with non-functional aspect related codes, are amenable to be benchmarked over the specific QoS parameters [Brah02] in the system deployment time.

4. Summary and Future Work

We have presented a two-level approach for handling the crosscutting concerns of functional/non-functional concerns in integrating heterogeneous distributed components. This approach has a close tie to MDA in the sense that we leverage component modeling at the first level and then map the component models into the CDL in the second level. The CDL and ASL will be weaved together to generate the executable specification for system simulation. The approach also applies to model weaving in MDA.

We have applied modeling techniques for enriching semantics of Web Services and to generate semantically enriched Web Service Description Language (WSDL) [Cao03]. We have also prototyped CDL for component assembly [Cao02]. Future efforts will be to apply modeling experiences to describing the semantics of component cases of some specific domain, and to build ASL together with its associated weaver for the synthesis of executable specifications.

References:

[Augu95] M. Auguston. Program Behavior Model Based on Event Grammar and its Application for Debugging Automation. *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, pp. 277-291, 1995.

[Brah02] G. J. Brahnmath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, and C. C. Burt. A Quality of Service Catalog for Software Components. *Proceedings of (SE)^2 2002, the Southeastern Software Engineering Conference*, pp. 513-520, 2002.

[11]DMS: Design Maintain System™, *http://www.semdesigns.com/*

[Brya02] B. R. Bryant, B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proceedings of 35th Hawaii Int. Conf. System Sciences*, 2002, http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf.

[Cao02] F. Cao, B. R. Bryant, R. R. Raje, M. Auguston, A. M. Olson, C. C. Burt. Component Specification and Wrapper/Glue Code Generation with Two-Level Grammar using Domain Specific Knowledge. *Proceedings of 4th International Conference on Formal Engineering Methods (ICFEM'02)*, LNCS 2495, Springer-Verlag, pp. 103-107, 2002.

[Cao03] F. Cao, B. R. Bryant, C. C. Burt, J. G. Gray, R. R. Raje, A. M. Olson, M. Auguston. Modeling Web Services: Toward System Integration in UniFrame, to appear in *Proceedings of 7th World Conference on Integrated Design and Process Technology (IDPT'03)*, 2003.

[Czar00] K. Czarnecki, U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Ducl02] F. Duclos, J. Estublier, P. Morat. Describing and Using Non Functional Aspects in Component Based Applications. *Proceedings of Second International Conference on Aspect-Oriented Software Development, AOSD'02*, 2002.

[GME01] *GME 2000 User's Manual, Version 2.0*, ISIS, Vanderbilt University, 2001.

[Kicz97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, Springer-Verlag, pp. 220-242, 1997.

[Kicz01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold An Overview of AspectJ. *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, LNCS 2072, Springer-Verlag, pp.327-353, 2001.

[Lieb99] K. Lieberherr, D. Lorenz, M. Mezini. Programming with Aspectual Components. *Technical Report, NU-CCS-99-01*, 1999, http://www.ccs.neu.edu/research/demeter/papers/aspectual-comps/aspectual.ps.

[OMG01] Object Management Group (OMG). Model Driven Architecture: A Technical Perspective. *Technical Report. Document # ormsc/2001-070-1*, Framingham, MA, Object Management Group, 2001.

[Raje00] R. R. Raje. UMM: Unified Meta-object Model for Open Distributed Systems. *Proceedings of ICA3PP, 4th IEEE Int. Conf. Algorithms and Architecture for Parallel Processing*, pp. 454-465, 2001.

[Raje01] R. R. Raje, B. R. Bryant, M. Auguston, A. M. Olson, C. C. Burt. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. *Proceedings of Monterey Workshop Engineering Automation for Software Intensive System Integration*, pp. 109-119, 2001.

[Shaw96] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[Stei02] D. Stein, S. Hanenberg and R. Unland. On Representing Join Points in the UML. *Aspect Modeling with UML Workshop at the Fifth International Conference on the Unified Modeling Language and its Applications*, 2002, http://www-stud.uni-essen.de/~sw0136/wissensArbeiten/UML02Workshop.pdf.

[Zhao03] W. Zhao, B. R. Bryant, C. C. Burt, J. G. Gray, R. R. Raje, A. M. Olson, M. Auguston. A Generative and Model Driven Framework for Automated Software Product Generation. *Proceedings of CBSE 6, the 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, 2003, http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6/Proceedings/papersfinal/p31.pdf.

Integrated Design and Process Technology, **IDPT-2003**
Printed in the United States of America, June, 2003
©2003 Society for Design and Process Science

MODELING WEB SERVICES: TOWARD SYSTEM INTEGRATION IN UNIFRAME

Fei Cao, Barrett R. Bryant, Carol C. Burt, Jeffrey G. Gray
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294, USA
{caof, bryant, cburt, gray}@cis.uab.edu

Rajeev R. Raje, Andrew M. Olson
Department of Computer and Information Science
Indiana University Purdue University at Indianapolis
{rraje, aolson}@cs.iupui.edu

Mikhail Auguston
Computer Science Department
Naval Postgraduate School
auguston@cs.nps.navy.mil

ABSTRACT

Web Services offer a platform independent solution for system integration in a distributed environment. But Web Services are weak in representing the business semantics of application domains. This paper presents a model-driven approach for specifying domain-specific component models in an effort to complement the current Web Services technology in terms of enriching the semantics representation. Web Services Description Language (WSDL) can then be generated automatically from the models with generators. The modeling of domain-specific components serves as a front-end to represent the semantics of components as well as for formalizing components while the generated artifacts facilitate component service synthesis.

1. Introduction

The integration and reuse of legacy software systems offer a promising direction for boosting productivity by dramatically reducing both cost and time-to-market expenses. One of the Object Management Group (OMG) initiatives is Model Driven Architecture (MDA)[1], in which legacy systems and Commercial-Off-The-Shelf (COTS) software can be transformed by reverse engineering into Platform Independent Models (PIMs) representing business functionality with underlying technical details presented abstractly. If this effort is successful, legacy systems and COTS software can be reintegrated into new platforms efficiently and cost-effectively. But for legacy systems and COTS software, the business logic and the software structures are usually encapsulated as black boxes, which makes it difficult to be reverse engineered. Hence, it is necessary to include the design artifacts (such as models, high-level specifications, etc.) in the business components. To that end, the vision of MDA also includes packaging models together with parameterized generators. The application generator will produce customized components according to the configuration parameters. In that way, not only can the footprint of business systems be minimized, but also various kinds of artifacts of business system can be generated on demand for system synthesis.

On the other hand, Web Services (WS)[2] technology offers a platform-independent solution for Enterprise Application Integration (EAI) by wrapping legacy systems as WS [Grah02]. Combining the model-driven approach with WS technology, software systems can be produced by synthesizing distributed models using generator technology.

UniFrame [Raje01] is a framework for seamless integration of heterogeneous distributed software components to assemble a complete distributed software system. The assembly process involves the generation of glue/wrapper code [Brya02], which is a challenging ad-hoc task considering the heterogeneous nature of distributed components. Because WS are based on open industry standards working across different platforms, wrapping heterogeneous components with WS for integration will transform the assembly task from n*m to n*1 processes (see Figure 1). The contribution of this paper is to propose the use of WS as a potential vehicle for system integration in UniFrame by enhancing semantic expressive power of WS using the model-driven approach. The related process is described herein.

In this paper, we present an approach based upon the principle of Model-Integrated Computing (MIC) [Léde01] to model the business domain-specific UMM component models. This involves a graphical modeling

[1] http://www.omg.org/mda/

[2] http://www.w3.org/2002/ws/

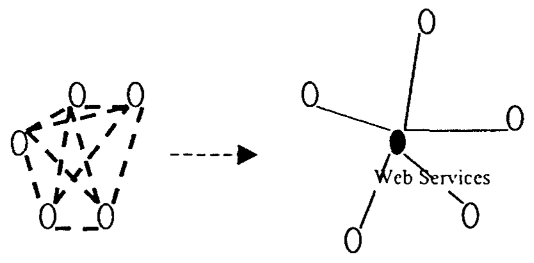


Figure 1. Reducing Gluing/Wrapping Process

environment for customizing a domain system based on domain-specific meta-models. An interpreter is built to generate WS Description Language (WSDL)[1] for business service integration. A generator can also be created to directly synthesize the implementation code. Figure 2 gives an overview of the approach.

This paper is organized as follows. Section 2 introduces the background knowledge of the UniFrame project, for which the proper meta-model development is imperative. Section 3 introduces the modeling environment and modeling targets with regard to the UMM model and WS. Section 4 describes the interpreter that generates the WSDL. A banking example is given in Section 5 illustrating the proposed approach. This paper ends with the conclusions and outlook in Section 6.

2. UniFrame

UniFrame is based on the Unified Meta-Component Model (UMM) [Raje00] for describing components. Systems constructed by component composition should meet both functional and non-functional requirements such as Quality of Service (QoS) requirements [Raje02]. UniFrame includes a specification of appropriate QoS parameters, which provide metrics of service at both the component level and system level, so that the software system produced by assembling heterogeneous components can be benchmarked over not only functional requirements, but also non-functional criteria. A Generative Domain Model (GDM) [Czar00] is used to describe the properties of domain- specific components and to elicit the rules for component assembly.

2.1 UMM

In the Unified Meta-Component Model (UMM), we are concerned about the following three aspects:

a) Component:

In [Medv97], components are described as being composed of the following aspects: interface, types, semantics, constraints and evolutions. But, this view does not reflect the collaborative features of distributed components. We believe that a component, as a provider for computational functionality and a gateway for further resource offerings, has not only computational aspects, but also cooperative aspects in distributed environments, as well as other auxiliary aspects like mobility and security.

b) Service and Service Guarantees.

Here we are focusing on providing metrics for quantifying the services provided by components as a criteria for making choices from multiple service providers, as well as criteria of judging assembled system by composing components. Once a component does not satisfy the expected QoS, it is a candidate for substitution. By modeling QoS aspects in the meta-model, we can weave the QoS instrumentation into generated code for QoS measurements at deployment time.

c) Infrastructure

In UniFrame, the Internet Component Broker (ICB) and Headhunters [Sira02] are proposed as two facilities in an effort to seamlessly integrate heterogeneous components. ICB provides translation capacity in terms of adapter technology for achieving interoperability, while Headhunters actively detect the presence of new components in the search space, register their functionality and attempt match-making between client components (service requesters) and server components (service providers). By generating such component specifications in XML, a component can be exposed for external querying, e.g., using XQuery[2]. Also, a pre-built meta-model, from which the domain-specific model is created, represents the domain ontology [Grub93] and provides the leverage for the ICB and Headhunter.

The aforementioned three concerns necessitate a proper methodology of creating a meta-model to modeling the following categories:

Table 1. Component Description in UMM

Computational Attributes	Inherent Attributes	ID
	Functional Attributes	Description
		Algorithm
		Complexity
		Syntactic Contract
		Technology
		...
Cooperation Attributes	Precondition	
	Postcondition	
Auxiliary Attributes	Security	
	Mobility	
	....	
QoS Metrics	Availability	
	End-to-End delay	
	......	

[1] http://www.w3.org/2002/ws/

[2] http://www.w3.org/XML/Query

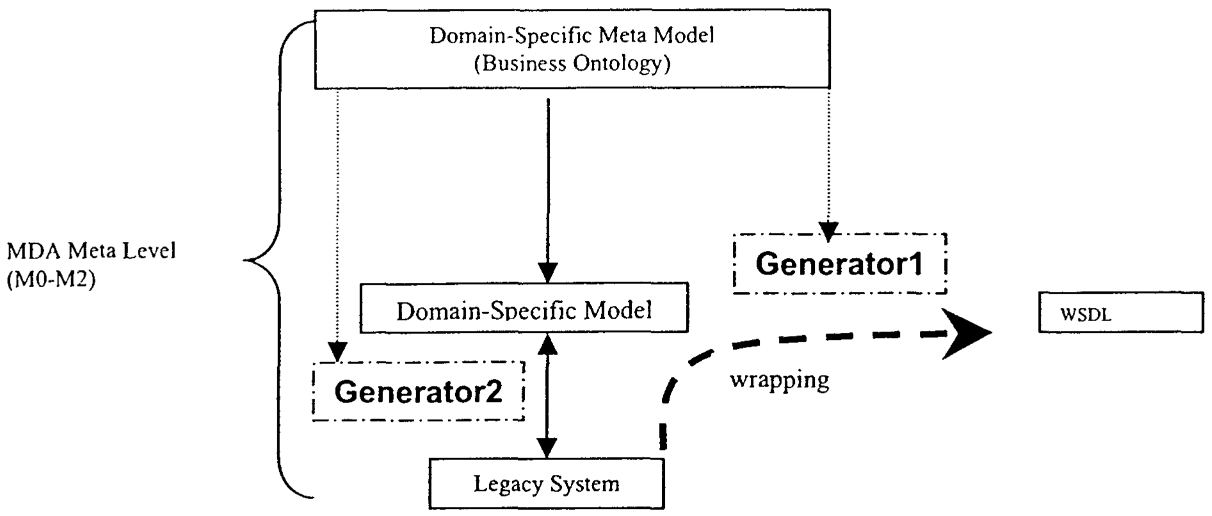


Figure 2. Overview of Approach

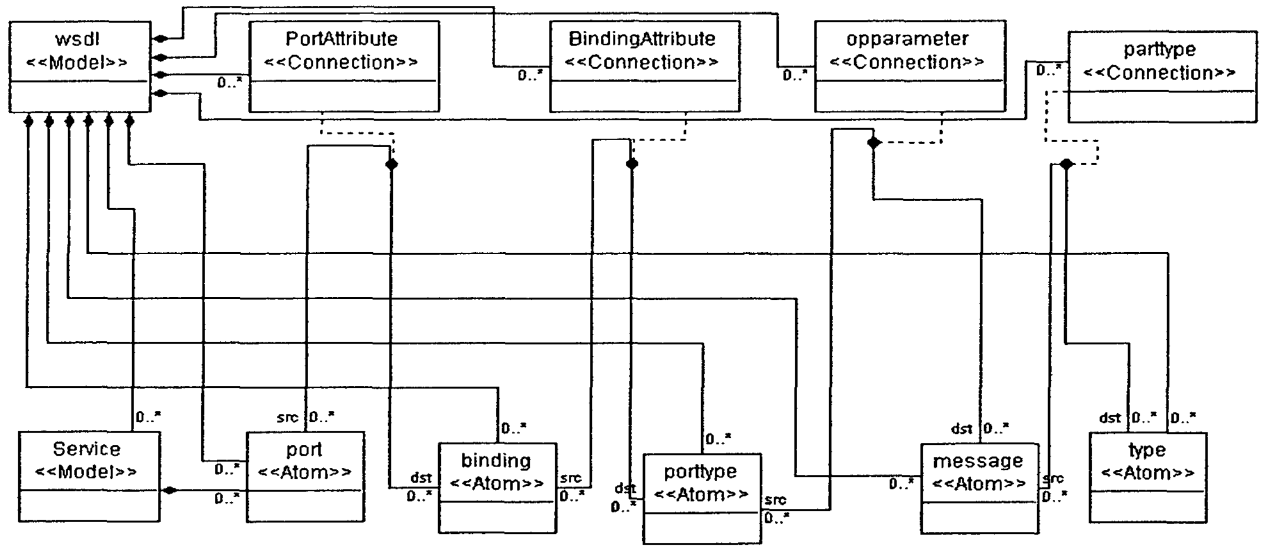


Figure 3. Meta-Model of WSDL

Obviously, a pure textual specification of UMM, while still a viable choice, will be error prone and hard to be processed and reused. The widely used Rational Rose [Quat00] toolkits, however, can only be used for non-executable modeling, in the sense that you have no control over generation of complete applications, which is not adequate enough for modeling UMM. This problem will be addressed using tool support introduced in the next section.

3. Modeling as the Front End of Web Services

3.1 Generic Modeling Environment (GME)

Model Integrated Computing (MIC) employs meta-modeling to define the domain modeling language and model integrity constraints. It uses these meta-models to automatically compose a domain-specific design environment and generate input to some analysis tools such as Matlab Simulink/Stateflow [Neem02]. MIC includes the Generic Modeling Environment (GME) for creation of domain-specific models, a Model Database for model storage, and a Model Interpretation technology for building model interpreters. In GME, the meta-models use Unified Modeling Language (UML) class diagrams to model the system information. Figure 3 gives the WSDL meta-model using GME. Also MCL (MGA[1] Constraint Language) [GME00], which is a subset of UML OCL[2] with some MGA specific extension, is used to enforce some

[1] MultiGraph Architecture [Szti95]

[2] http://www-3.ibm.com/software/ad/library/standards/ocl.html

semantic rules in MGA modeling paradigms. This adds some formalism to the modeling, which can be used to enrich the semantic expressiveness of WSDL, as is explained later in section 5

WSDL is not convenient to be manually coded. Many tools such as AXIS[1], and the Microsoft .Net framework provide the function of generating WSDL from implementation code (such as Java and C#) and vice versa. Such tools leverage compiler technology to generate WSDL from some other programming languages. In contrast, by generating WSDL from a high-level language-independent model, we can avoid the need for language-specific compilers. This permits easier maneuvering of the generated WSDL at a higher level. Also, by standardizing the meta-model and the associated generator, the domain ontology will be uniformly embodied in generated WSDL. This will facilitate program-to-program interoperation bearing the intelligence of software agents, such as autonomy and knowledge [Gris01].

3.2 Enriching and Modeling WS Semantics

Current WS standards mainly embody the semantics of processes at the collaborating syntactic interface level. WSDL only exposes distributed object services, while such process behavior aspects as ordering, and dependency are not well specified in the existing WSDL standard. Figure 4 gives the meta-model of a Finite State Machine (FSM), which can be used to model the dynamic behavior of WS, in particular, the sequence of states that the WS behavior goes through in its lifetime. We will illustrate this point in detail in a later example.

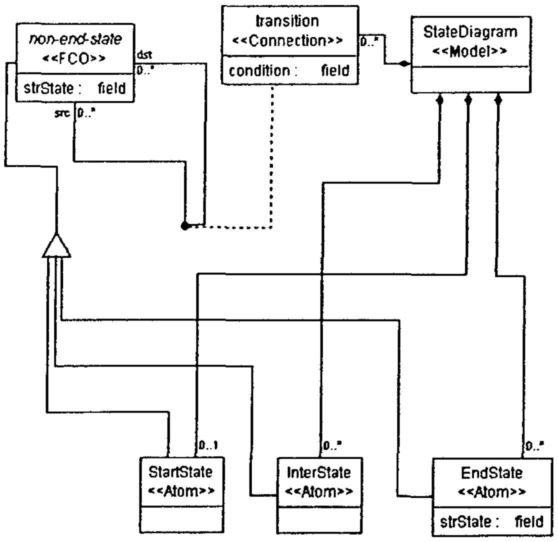


Figure 4. Finite State Machine (FSM) Meta-model

[1] http://ws.apache.org/axis/

4. Web Services Generator

A key aspect of MDA is the generator technology. By generating implementation code from a high-level specification language, software systems can be produced with high efficiency while the scale of software reuse will be reduced at the specification level. GME provides the Builder Object Network (BON) framework [GME00] for building interpreters by instantiating each object in the model tree with a C++ object. The objects in the model tree can be traversed by calling methods within the BON API. In order to precisely generate target code from the models using a generator, a special *atom* can be added in the GME environment denoting specific meaning so as to enrich the semantics of modeling. e.g., in feature modeling [Czar00], there are mandatory features, optional features and alternative features for some concept. We can add a *Require* atom, an *Or* atom, an *XOR* atom to denote the three relationships between other atoms. Figure 5 illustrates the strategy. In this way, the designated semantics can be captured when traversing the model tree. This strategy can also be applied to model UML relationships such as *Dependency, Generalization, and Association.* In this way, the built-in class diagram facilities of GME itself can be extended.

5. Putting it Together

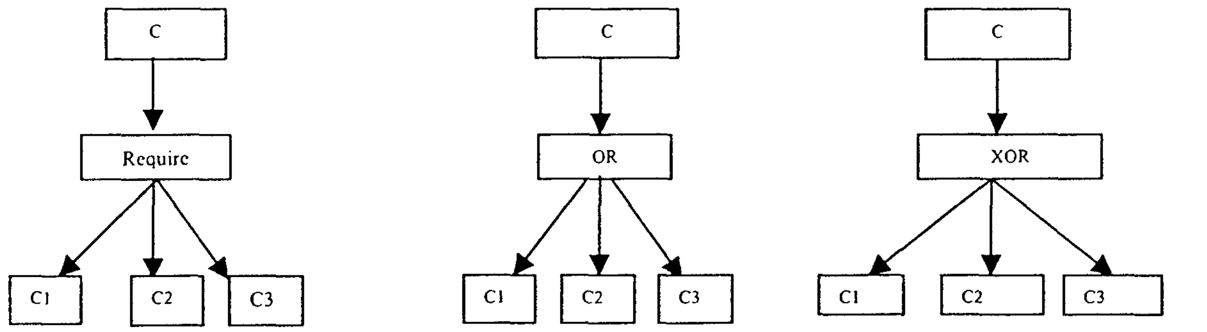
This section will use GME to create a meta-model embracing both UMM and WS, and an interpreter is built based on this meta-model for generating WSDL in an effort to facilitate component service synthesis in UniFrame.

5.1 Creating Banking Domain Meta-Model

Below is a simple banking domain specification:

A bank provides the service for users to set up accounts. Account information includes personal data including Name, SSN, phone number, address, and account data including Account Number, PIN, Transaction Record, Balance. There are two types of accounts: checking account and savings account.

For the bank side, it provides such services as: Account Validation (to ensure legal access of account), Account Verification (to double check the account after each transaction, including transaction history, transaction description, etc), Account Query (balance checking), Deposit, Withdraw, and Transfer. There is order restriction for those operations. Both Transfer and Withdraw have to be preceded by a Query operation. The Account Verification comes after each of the other operations.



(1) Concept C requires all the three features: C1, C2, C3

(2) Concept C contains some of the three features: C1, C2, C3

(3) Concept C contains exactly one of the three features: C1, C2, C3

Figure 5. Representing Semantics of Feature Modeling with Atom-to-Atom Connection

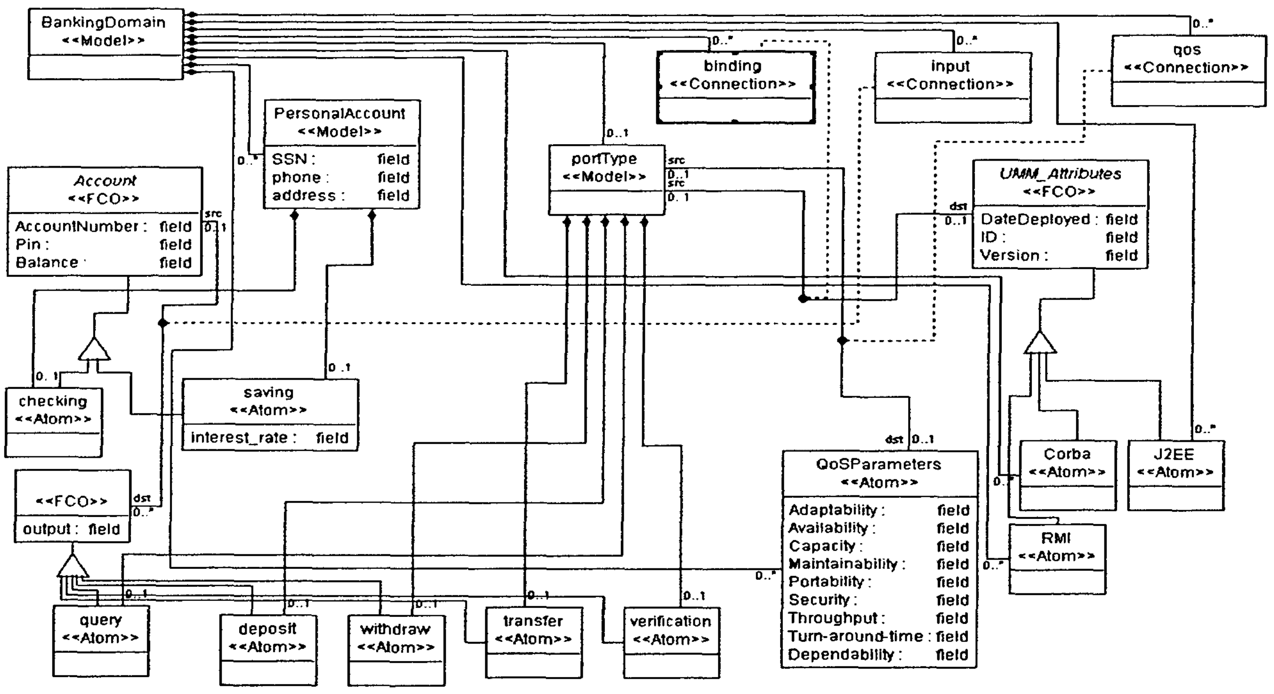


Figure 6. Meta-model of Banking Domain

Deposit and withdraw can only be applied to checking account (this is not the generic case, though). The aforementioned services are optional so long as the above rules are observed.

The banking service may leverage such technology as RMI, J2EE, and CORBA. Also it will enforce some QoS concerns such as Availability, Dependability, Capacity, etc. (For more QoS parameters see [Raje02]).

Directly expressing the above specification in WSDL will tend to blur the 4+1 view[1] of the software architecture. Thus it is hard to represent the intended requirements precisely and the constraints can not be warranted. Model-based WSDL generation will be able to solve the ambiguity problem by clearly modeling the specification in a graphical fashion to capture all the involved relationships. The meta-model in Figure 6 represents the banking domain knowledge. It's derived from WSDL elements and banking domain knowledge. *portType* in WSDL denotes the WS abstract interface definition. It is represented as a *model* in Figure 6, which contains the following banking-domain specific operations: *query, deposit, withdraw, transfer, verification. binding* in WSDL denotes how the elements in an abstract interface (*portType*) are converted into a concrete representation in a particular combination of data formats and protocols (here, platform specific implementation in

[1] which includes functional requirements, software module organization, run-time implementation structure of the system, etc. For details see [Kruc95]

CORBA, J2EE, RMI, etc). Consequently, *binding* is represented as a *connection* between *portType* and *UMM_Attibutes*, which is the parent of the CORBA, J2EE and RMI *atoms*.

The left part of Figure 6 (*PersonalAccount*, *Account*, *checking*, *saving*) is basically about a simplified version of the feature modeling [Czar00] of the banking domain, which is treated as input (represented as *connection* here) into operations of *portType*. Also QoS parameters, by being associated with *portType*, will be embedded into the generated WSDL as extended attributes. WSDL itself is XML based, so a query expressed in XQuery can make use of extended WSDL attributes to refine the query in selecting targeted WSDL. Here, the listed QoS parameters are treated as of static type. For dynamic parameters, we can apply aspect weaving [Kicz97] technology in the code generation phase for performing dynamic measurements.

The specified constraints over *withdraw* and *deposit* operations can be enforced in GME using the following MCL (refer back to section 3.1) expression:

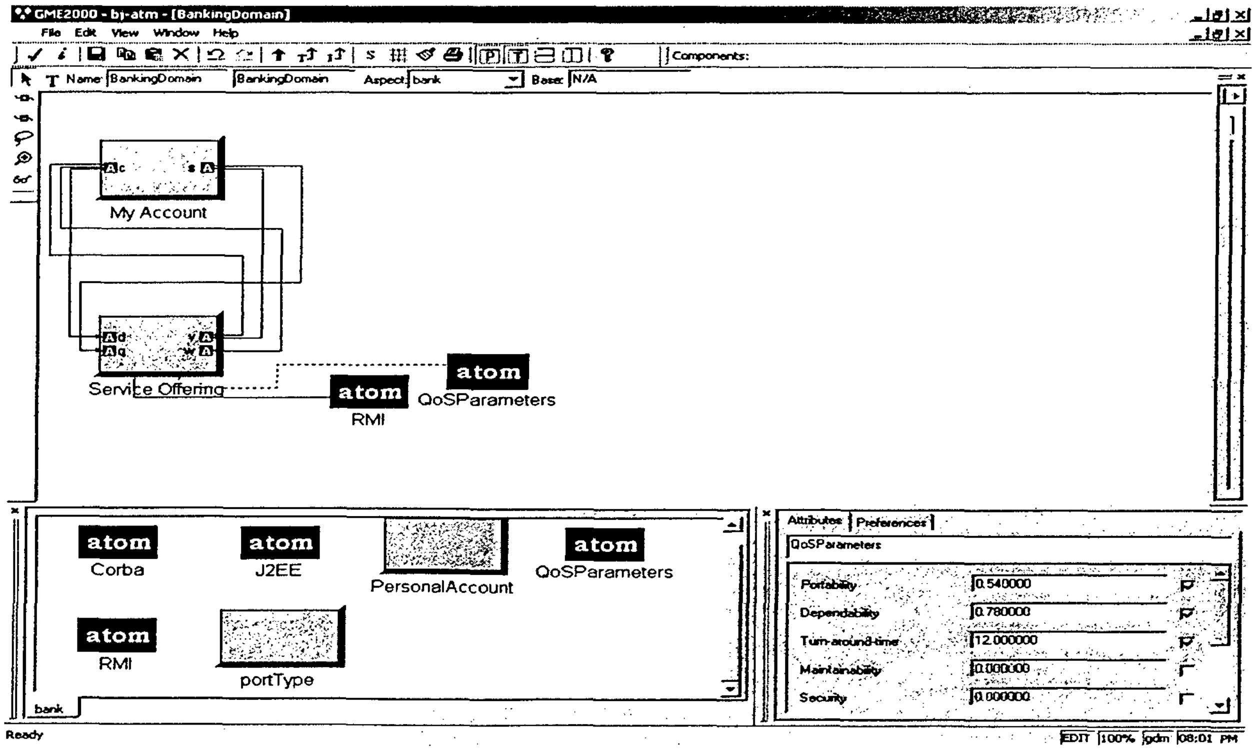
```
connectedFCOs("src")->forAll(
         c|c. kindName() ="checking")
```

Those constraints apply to both the *withdraw* atom and the *deposit* atom in Figure 6, which means those First Class Objects (FCO: referring to both entities and relations in GME) that are connected with *withdraw/deposit* atoms are all of kind "checking"; i.e., those services can only be applied to *checking account*.

But, when it comes to the handling of order constraints as specified in the banking domain example, obviously MCL is not adequate enough to capture such dynamic behaviors. Such modeling techniques as using the Finite State Machine will provide modeling capacity for advanced behavior, which is detailed in the next section.

5.2 A Banking Model and WS-based Integration

Figure 7 is an example of the banking model. For this model, "My Account" is the name for the "PersonalAccount" model. It has two kinds of account: both checking (c) and savings (s). "Service Offering" represents the "portType". It offers 4 types of service (without transfer in this case): d: *deposit*, q: *query*, w: *withdraw*, v: *verification*. From the connections between the ports we can see for this banking model, the *query* can only be applied to the savings account, while *verification* can be carried out over both types of account. *Withdraw* and *deposit* only applies to checking account. Otherwise the modeling environment will give warnings when modeling, which is consistent with the MCL specification. Also, notice for this banking model, RMI technology is adopted and some QoS parameters are specified here, as shown in the lower-right corner attribute list. The attribute list associated with RMI will also be shown in the corner if the RMI atom is under focus.



QoSParameters

Portability	0.540000	✓
Dependability	0.780000	✓
Turn-around-time	12.000000	✓
Maintainability	0.000000	
Security	0.000000	

Figure 7. "My Account": a Banking Model

From the model in Figure 7 the interpreter will generate two sets of codes: the WSDL code for the banking service embedded with QoS parameter extension, and the WS wrapping code for the underlying RMI implementation. Because the generated WSDL is quite lengthy, we will just show some model-specific contents as shown in the following paragraph. Notice the bold-font part of the following WSDL represents the QoS extension of WSDL, which may be used for WS filtering if QoS requirements are submitted in the query expression.

```
<definition name="my bank">

<types>
  <xsd:schema
   targetedNamespace="http://localhost/bank"
   xmlns:xsd="http://www.w3
       .org/2001/XMLSchema">
  <xsd:complexType name="Account">
      <xsd:sequence>
         <xsd:element  name="AccountNumber"
           type="xsd:string"/>
          <xsd:element name="Pin"
              type="xsd:string"/>
          <xsd:element name="Balance"
              type="xsd:decimal"/>
      </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="checking">
      <xsd:complexContent>
          <xsd:extension  base="Account">
      </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="savings">
   <xsd:complexContent>
          <xsd:extension base="Account">
    <xsd:attribute name="interest_rate"
     type="xsd:decimal"/>
   </xsd:complexContent>
  </xsd:complexType>
 </xsd:schema>
</types>

<message name="checking">
 <part name="p1" type="checking"/>
</message>
<message name="savings">
 <part name="p1" type="savings"/>
</message>
<message name="checking_savings">
 <part name="p1" type="checking"/>
 <part name="p2" type="savings"/>
</message>

<portType name="bankPortType">
    <operation name="withdraw">
        <input message="checking"/>
       <output message=""/>
    </operation>
  <operation name="deposit">
       <input message="checking"/>
       <output message=""/>
  </operation>
 <operation name="verification">
  <input message="checking_savings"/>
  <output message=""/>
 </operation>
 <operation name="query">
    <input message="savings"/>
     <output message=""/>
 </operation>
</portType>
```

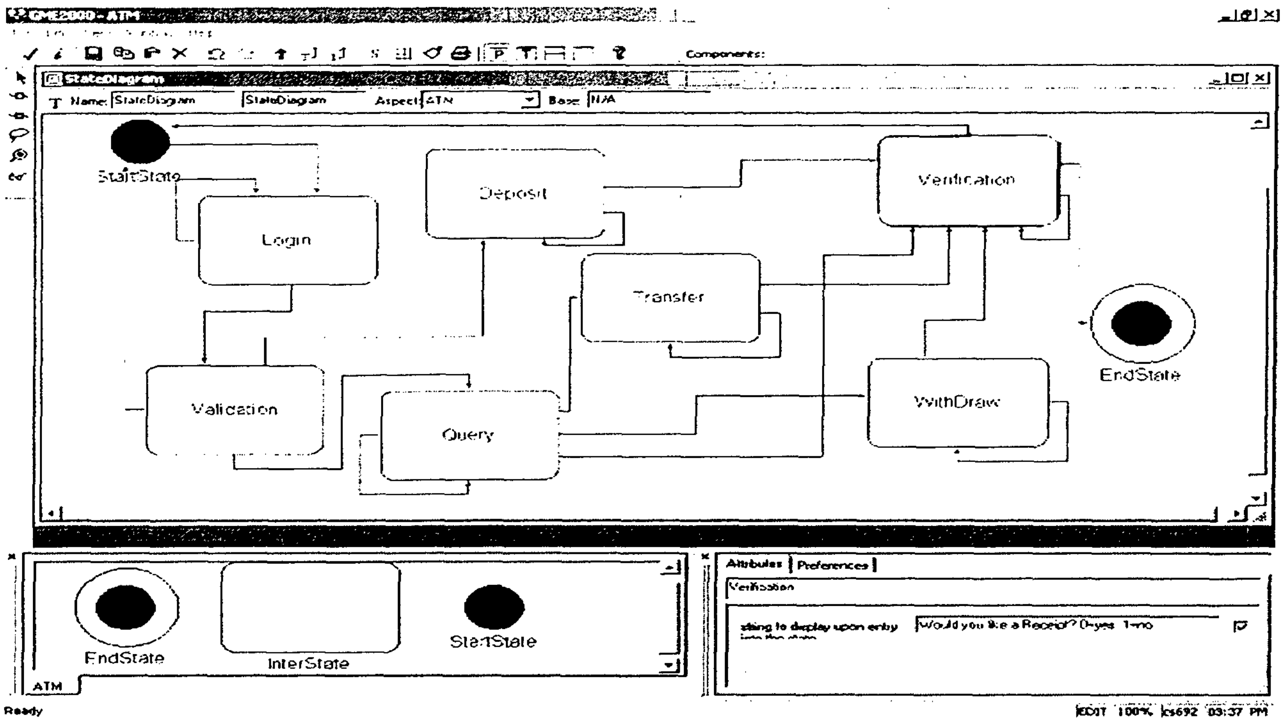
```
<binding>
.........
</binding>

<service name="My Bank" Portability="0.544400"
   Dependability="0.780000" Turn-around-
   time="12.000000"/>
  <port>
 .....
  </port>
</service>

</definition>
```

Now we turn to the handling of the order restriction requirement in the banking domain specification. We will use the FSM meta-model (Figure 4) to build the banking service state model as shown in Figure 8 and the associated interpreter. Because every service corresponds to the child node (atom) of *portType model* in Figure 6, we can use BON API (refer back to Section 4) to traverse those child *atoms* of *portType* in the banking model one by one while retrieving the connection information of each *atom*. The generated WSDL extension describing the state transition process is as follows:

```
<state>
    <state name= "Login" >
    <state name="Validation" >
    <state name="Query" >
    <state name="Deposit" >
    <state name="Transfer" >
    <state name="Withdraw" >
    <state name="Verification" >
</state>
<transition>
  <transition src="StartState"
   dst="Login" condition="">
  <transition src="Login" dst="Login"
   condition="">
  <transition src="Login"
   dst="Validation" condition="">
  <transition src="Validation"
    dst="Deposit" condition="">
  <transition src="Validation"
   dst="Query" condition="">
 <transition src="Deposit" dst="Deposit"
   condition="">
  <transition src="Deposit"
    dst="Verification" condition="">
  <transition src="Query" dst="Transfer"
     condition="">
  <transition src="Query" dst="Query"
    condition="">
  <transition src="Query" dst="Withdraw"
    condition="">
  <transition src="Query"
    dst="Verification" condition="">
  <transition src="Transfer"
    dst="Transfer" condition="">
  <transition src="Transfer"
    dst="Verification" condition="">
  <transition src="Verification"
    dst="StartState" condition="">
  <transition src="Verification"
     dst="Verification" condition="">
  <transition src="Verification"
    dst="EndState" condition="">
  <transition src="WithDraw"
   dst="WithDraw"  condition="">
  <transition src="WithDraw"
    dst="Verification" condition="">
</transition>
```



Verification

string to display upon entry into the state: Would you like a Receipt? 0-yes 1-no

Figure 8. Banking behavior model based on FSM meta-model

Note in the generated state transition code, the "condition" attributes are supposed to be customized in the specific banking behavior model before code generation, which for the sake of brevity are left blank here. The state transition specification generated here may be used in guiding the WS consumption and composition.

6. Conclusions and Future Research

This paper applies the model driven approach to WS technology. By modeling service behavior at a higher level, the system semantics can captured at a finer grain. Meanwhile, different artifacts can be derived from models using a generator, which will not only refine the service presentation, but also facilitate system integration. In particular, this approach is applied in the context of the UniFrame project for system integration. So far, we have implemented a prototype with the function of WSDL generation from a specific component model and FSM modeling for component services.

Because the meta-model is the starting point and cornerstone of system integration, we will need to refine the meta-model leveraging domain knowledge until it can be standardized. To enhance the semantics expressing capability of WS, future research will involve not only state machine modeling, but also the modeling of other behavior concerns, such as interaction, activity, process/thread and temporal relationship. Also, technology and QoS modeling in the above banking example are still quite primitive, both of which need further exploration for the ultimate model-based glue/wrapper code generation between WS and other component models.

Acknowledgements. This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

REFERENCES

[Brya02] Bryant, B. R., Auguston, M., Raje , R. R., Burt, C. C., Olson , A. M., 2002, "Formal Specification of Generative Component Assembly Using Two-Level Grammar," *Proc. SEKE, 14th Int. Conf. Software Engineering and Knowledge Engineering,* pp. 209-212.

[Czar00] Czarnecki, K., Eisenecker, U.W., 2000, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley.

[GME00] "GME 2000 User's Manual, Version 2.0," 2001, ISIS, Vanderbilt University.

[Grah02] Graham, S., Simeonov, S., Boubez, T., Davis, D., Daniels, G., Nakamura,Y., Neyama, R., 2002, *Building Web Services with Java*, SAMS.

[Gris01] Griss, M., 2001, "Software Agents as Next Generation Software Components", *Component-Based Software Engineering*, ed. Heineman, G. T., Councill, W. T., Addison-Wesley, pp. 641-657.

[Grub93] Gruber, T. R., 1993, "A translation approach to portable ontology specifications," *Knowledge Acquisition,* Vol. 5, No. 2, pp.

199-220.

[Kicz97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J., 1997, "Aspect-Oriented Programming," *Proc. ECOOP, European Conference on Object-Oriented Programming*, Springer-Verlag LNCS Vol. 1241, pp. 220-242.

[Kruc95] Kruchten, P.B., 1995, "The 4+1 Views Model of Architecture", *IEEE Software*, Vol. 12, No. 6, pp. 42-50.

[Léde01] Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J. and. Karsai, G., 2001, "Composing Domain-Specific Design Environments," *IEEE Computer*, Vol. 34, No. 11, pp. 44-51.

[Medv97]Medvidovic, N., Taylor, R.N., 1997, "A Framework for Classifying and Comparing Software Architecture Description Languages, " *Proc. ESEC/FSE '9, European Software Engineering Conf./9th Conf. Foundations of Software Engineering*, Springer-Verlag LNCS Vol. 1301.

[Neem02] Neema, S., Bapty, T., Gray, J., Gokhale, A., 2002, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," *Proc. GPCE, First ACM SIGPLAN/SIGSOFT Conf. Generative Programming and Component Engineering*, Springer-Verlag LNCS Vol. 2487, pp. 236-251.

[Quat00] Quatrani, T., 2000, *Visual Modeling with Rational Rose 2000 and UML*, Addison Wesley.

[Raje00] Raje, R., 2000, "UMM: Unified Meta-object Model for Open Distributed Systems," *Proc. ICA3PP, 4th IEEE Int. Conf. Algorithms and Architecture for Parallel Processing*, pp. 454-465.

[Raje01] Raje, R., Bryant, B., Auguston, M., Olson, A., Burt, C., 2001, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," *Proc. Monterey Workshop Engineering Automation for Software Intensive System Integration*, pp. 109-119.

[Raje02] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., Burt, C. C., 2002, "A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components," *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 2, pp. 1009-1034.

[Sira02] Siram, N. N., Raje, R. R., Olson, A. M., Bryant, B. R., Burt, C. C., Auguston, M., 2002, "An Architecture for the UniFrame Resource Discovery Service," *Proc. SEM, 3rd Int. Workshop Software Engineering and Middleware,* Springer-Verlag LNCS Vol. 2596.

[Szti95] Sztipanovits, J., Karsai, G., Biegl, C., Bapty, T., Lédeczi, Á., Misra, A., 1995, "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proc. IEEE ICECCS, International Conference on Engineering of Complex Computer Systems*, pp. 361-368.

Automated Glue/Wrapper Code Generation in Integration of Distributed and Heterogeneous Software Components

Wei Zhao, Barrett R. Bryant, Fei Cao, Carol C. Burt
*Computer and Information Sciences Department
University of Alabama at Birmingham
Birmingham, AL 35294-1170, U.S.A.
{zhaow,bryant,cburt}@cis.uab.edu*

Rajeev R. Raje, Andrew M. Olson
*Computer and Information Science Department
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202, U.S.A.
{rraje, aolson}@cs.iupui.edu*

Mikhail Auguston
*Computer Science Department
Naval Postgraduate School
Monterey, CA 93943, USA
auguston@cs.nps.navy.mil*

Abstract

UniFrame is a framework to help organizations to build interoperable distributed computing systems. Using UniFrame, a new system is built by assembling pre-developed heterogeneous and distributed software components. UniFrame solves the heterogeneity problem by explicitly modeling the domain knowledge of various technology domains (component model domains, programming language domains, operating system platform domains, etc.), from which the Interoperation Generative Domain Model (IGDM) straddling the technology domains can be constructed. The glue/wrapper code that realizes the interoperation among the distributed and heterogeneous software components can be generated from the IGDM. In this paper, an informal implementation in Java of glue/wrapper code generator is given, followed by a discussion on a formalization of IGDM. The formalism comes from the fact that if the family of glue/wrapper code can be modeled formally, an instance glue/wrapper code can be generated automatically. In this formalization, the IGDM is formally modeled as a language definition using a grammar; the code that realizes the interoperation is a valid sentence derivable from the grammar, and will be generated automatically from the IGDM during the assembly time.

1. Introduction

In today's world, distributed computing systems (DCS) are omnipresent. The successes of organizations will largely depend upon their abilities to create robust and effective software for DCS. Despite the achievements of component-based software engineering in distributed computing environments, the inherent complexity, de-centralization and heterogeneity of DCS still remain risks and challenges. Achieving a seamless interoperation among heterogeneous distributed components would be the most critical task of building a successful DCS. UniFrame [Raj01], [Raj02] is such a framework to help organizations to build interoperable DCS.

To meet the challenges, UniFrame has the following three specific goals:

1. The genetic diversity and complexity of the world (a plethora of component models, programming languages, operating systems, communication protocols) causes separation and isolation among the technology islands. UniFrame provides a unified interoperation among the collaborating components.
2. The rapid technology evolution makes the application integration a real challenge. With the interoperability, the legacy features can be integrated into the system developed in new technologies.
3. The advances in the processor and networking technologies have changed the computing paradigm from a centralized to a distributed one. "The network is the computer." The ability to deal with distribution is essential to develop large scale DCS.

In short, UniFrame aims at the distribution and interoperation. Using UniFrame, a new system is built by assembling pre-developed heterogeneous and distributed software components. This paper will discuss the interoperation framework in UniFrame.

The paper is organized as follows. Section 2 distills some aspects of UniFrame that are relevant to the discussion of the interoperation framework. The interoperation framework is presented in section 3 with two alternative implementations (informal and formal). Some representative related work is given in section 4. The paper concludes in section 5.

2. Overview of UniFrame

Before we detail the interoperation framework, we first introduce the basics of the UniFrame.

2.1. Fundamental Theses of this Framework

Modularity and component-based software engineering. Component-based Software Engineering (CBSE) and related technologies have demonstrated their strength in recent years by increasing development productivity and parts reuse. The implementation of UniFrame is built upon the maturity of component-based software engineering [Szy02]. In our framework, features are standardized domain services. They are the smallest and the most abstract units for reuse and re-construction. One or more services are developed as a single component. Given all the possible elementary services for a business domain, a wide spectrum of systems can be generated by various combinations of services. Components are registered to the native registry in their domain for later discovery, composition and trading. Components are alive on the Internet, offering their services, QoS assurance and associated price. The separation of reusable feature (asset) development in the domain engineering and the product configuration using those assets in application engineering reflect the fundamental discipline of the separation of component development and component composition.

Software development paradigm shift: from single application development to system family development. System family engineering is also called Generative Programming [Cza00] and Product-line Engineering [Cle01], [SEI02], [Wei99] with the goal to automatically generate concrete software products from a domain-specification and reusable components. System family engineering has two levels: domain engineering and application engineering [Kan98]. Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets. Application engineering is the process of producing concrete systems using the reusable assets developed during domain engineering. In GP, a model of a family of products is called the Generative Domain Model (GDM). The major constitutes of a GDM are a feature model for modeling the commonality and variability among the products, a generator to generate a specific product based on the feature model specification, and the implementation of reusable components from which the product can be generated. This concept of paradigm shift is the core design of UniFrame as well as the interoperation framework in UniFrame.

Capture, formalize, model and reuse engineering knowledge. Any software system has domain-specific concepts and logic, a structure, and an implementation in concrete technologies. Decisions made on how to produce the software using those concepts comprise the engineering knowledge. In current software engineering practice (single system development), the engineering knowledge is scattered among: 1) the business executives, 2) the domain experts, 3) the software managers and engineers, and 4) the software developers. During the software production process, the decisions made by all these participants contribute respectively towards: 1) the goal of the system, 2) detailed business logic of the system, 3) specifications of software architecture and developers' role assignments, and 4) concrete software development by applying different programming languages and component-based technologies.

However, when we move the development paradigm to the product-line assembly, with the goal of manufacturing the concrete software products from the GDM automatically, the engineering knowledge specific to that end product must be captured, modeled and formally defined in a domain model to guide the automated manufacturing in the application-engineering phase.

The applicability of a domain is flexible. A domain is a set of current and future applications that share a set of common capabilities and data [Kan90]. Based on the principle of separation of concerns, we have encountered different categories of domains in the process of automated product generation [Zha02]:

1. Business domain: ontology for business concepts, logic and hierarchical structure.
2. Architecture domain: ontology for software architectural patterns, software parts' functionality, role and collaborations.
3. Technology domain: ontology for implementation technologies, such as component models, programming languages, security methods, and hardware platforms.

The principle of autonomy and separation of concerns naturally shapes the categorization of those three domains. Different dimensions of engineering knowledge are built and maintained by different group of people with different education background and talent set. This gives them the opportunity to be more productive and concentrate on the essence of their job. For example, architecture and technology domain builders are more likely to have computer science education than business domain developers.

2.2. The Structure of the UniFrame Framework

As shown in figure 1, there are two phases in UniFrame: domain-engineering and application-engineering. The domain-engineering phase simulates the domain development of three-dimensional domains (business domains [Zha04], architecture domains and technology domains). As part of the activity in business domains, designated programmers implement business domain features as software components with facilities of Model Driven Architecture (MDA) [Fra03]. Components are registered to native component model registries (e.g., RMI registry, CORBA naming services registry). Along with a natural hierarchy of business organizations, a set of available components for an application are not limited to reside on one computer, one network or one organization. They will be dispersed over the Internet. So, component searching is one of the major concerns in UniFrame. The UniFrame Resource Discovery Service (URDS) [Sir02] searches federated native component registries in the business domain for matched components. Domain level development provides the meta-data and reusable assets for the application engineering.

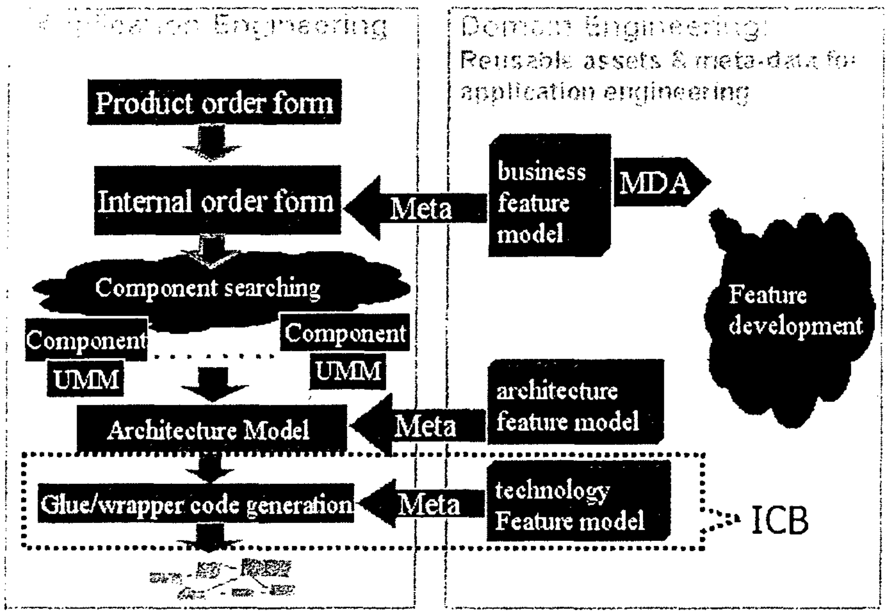


Figure 1. An Overview of the UniFrame Framework

The application-engineering phase is the process of manufacturing concrete products from the business domains. An order of a product is placed by using a user-friendly form such as HTML form, a GUI builder, a UML[1] model, a Generic Modeling Environment (GME) [GME] model or natural language. This order is translated into the internal representation that can be used for validation and initiating a search. We chose the XML for the internal representation. Then, the order is first validated according to the feature model in the business domain (no business logic violation [Zha04]). If this validation succeeds, URDS is invoked for searching the implementation components over the business domain space. When the URDS returns, a dummy composition of a set of candidate components is validated according to the feature model in the architecture domain (no architectural violation) with any necessary architectural instrumentation code generated automatically. Finally, if there are any incompatibilities in the component implementation technologies, the glue/wrapper code should be generated for the interoperation.

This paper will focus on the UniFrame interoperation framework that is called the Internet Component Broker (ICB), which is analogous to an Object Request Broker (ORB). As opposed to providing the capability to generate the glue and wrapper necessary for objects written in different programming languages to communicate transparently, the ICB provides the interoperation for

[1] Unified Modeling Language, http://www.uml.org/

components implemented in diverse component models and thus presents a collaboration vision one level above the ORB. For the interoperation of heterogeneous software components, ICB gives a vision of unified middleware.

2.3. Unified Meta-component Model (UMM)

Because of the separation of component implementation and component assembly, a unified component introspective mechanism is needed for the integration of components developed in diverse technologies. The Unified Meta-component Model (UMM) [Raj00] is such a mechanism that provides an abstraction for each component.

Our study has discovered that any individual feature implementation (component) reveals four aspects of knowledge in regards to the assembly process: computational, cooperative, deployment and economic aspects. As the domain grows, feature development would span multi-organization, multi-region/country, multi-time period, and multi-technology, which lends them a distributed and heterogeneous nature. UMM can formally and uniformly represent four aspects:

1. UMM computational aspects indicate implemented services, algorithms used, complexity, service contracts (component interface), service usage patterns. Parameters in UMM computational aspects identify features in the business domain.
2. Components are developed for reuse. UMM cooperative aspects take care of the interrelationship among the components, the individual functionality role contributing to the whole system, etc. Parameters in UMM cooperative aspects identify the entity and entity relationship in the architecture domain.
3. Some deployment issues such as component model and programming language used, operating system platforms, underlying network quality, CPU and memory usage, etc., constitute the deployment aspect of the UMM. UMM deployment aspects present the technology domain features for generating interoperation and deployment instrumentation code.
4. UMM economic aspects straddle business, architecture and technology domains, identifying the QoS parameters in each domain.

If the system assembly succeeds, a new UMM specification will be generated as well by composing component UMMs so that the new product can act as a reusable component for subsequent system generations.

There are several ways to develop UMM:

1. UMM is first documented in natural language, and then transformations can be applied to transform the informal UMM specification to formal models, and finally to the implementation software components [Bry03], [Lee02a], [Lee02b].
2. UMM is developed as a design model (e.g., UML) or a domain-specific model (e.g., GME), then a MDA approach is adopted to transform a business model to a Platform Specific Model (PSM) [Fra03], which will generate APIs, which will then be fine-tuned with concrete implementations.
3. Components are developed first, and then UMMs are generated from the implementation via some tool support.

Currently in our prototype, UMM is in a mix of natural language and XML, and can be generated from a Platform Independent Model (PIM) developed in GME [Cao03]. The components are developed manually by the programmer conforming to the feature specifications.

2.4. Quality of Service (QoS)

During component assembly, QoS is an important concern to ensure that the generated product meets the quality of service in the product order requirements. The QoS requirements are expressed by selecting an appropriate set of parameters from a catalog of QoS parameters [Bra02], [Raj02]. We have summarized and published 18 QoS parameters. QoS is business related (speed of the car, the aliveness of a supply chain), architecture related (structure integrity) or technology related (security level, turnaround time). QoS parameters are divided into two categories: a) static (the value can be obtained from UMM, such as encryption level), b) dynamic (the value can only be obtained from composition run-time, such as turnaround time). By using event grammar [Aug97], the dynamic QoS provides dynamic metrics that can be generated during the assembly time and be weaved into the glue/wrapper code. For example, we can use AspectJ[2] to weave in the turnaround time testing probe into the glue/wrapper code.

It is always possible that URDS will find multiple components with compatible static QoS, and so the dynamic QoS metrics will further refine the candidate set to generate a system that meets the user's QoS expectation of the final system.

[2]AspectJ project, *Eclipse.org*, http://www.eclipse.org/aspectj/index.html

3. Interoperation Framework in UniFrame

In this section, a detailed discussion of the interoperation framework in UniFrame is given followed by two alternative ways of implementation.

3.1. UniFrame Interoperation Framework

Potentially, there are several ways to establish the interoperation among the heterogeneous and distributed software components:

1. Source-to-source transformation: completely translate a component into the technology of its communicator. One example would be to use program transformation for legacy component migration [Bax04]. This type of technology is usually used during the reengineering [Ben87] of legacy systems. But source-to-source transformation can not be used as a general solution for the interoperation of heterogeneous software components because the complexity involved in establishing interoperation is $O(n^2)$. Considering there are n components, n(n-1)/2 transformations are needed for a full connected interoperation among n components. Despite the complexity, the source-to-source transformation is generally considered hard, and normally has to depend on a sophisticated commercial tool such as the Design Maintenance System [Bax04].
2. Transforming communicating components into a common technology for interoperation will significantly lower the interoperation complexity to O(n) since only n transformations are needed to transform n components into a common technology. An obvious example is using XML as an exchangeable technology for interoperation among different data forms.
3. Meta-interoperation is a specialization of the second item above. The common entity in meta-interoperation is not (or not only) the common technology used, but (also) is the meta-data for the transformation. Apparently, XML Meta-data Interchange (XMI) [Gro02] falls in this category, e.g., XMI defines a standard schema for object-XML mapping so that different objects can be mapped to a unified XML. MDA for the purpose of interoperation among different technologies is another example. MDA defines the standard mapping from a common Platform Independent Model (PIM) to different Platform Specific Models (PSMs) so that components in one PSM can interoperate with components in another PSM. CORBA [Vin97], [Corba] for interoperation among distributed components that are written in different programming languages also belongs to this category because the IDL can be considered as a PIM.
4. Three items listed above are all targeting translating the communicators. However, source-to-source semantic translation of software components, model (in the case of MDA), or APIs (in the case of CORBA), is laborious and error prone. The last possibility for interoperation is translating the communications instead of communicators. In terms of the size of the entity to be translated, the communication in general is magnitudes smaller than the communicators themselves. As a result, translating communication is the lightest way of establishing interoperation, which is usually realized by messaging. UniFrame has subscribed to this approach.

Before detailing the UniFrame interoperation framework, we first introduce the hypothesis we adopted. In the vision of UniFrame, components are autonomous and live in their own technology territory. In such territory, there is a central registry where components can be registered and be invoked from. Components, after being manufactured, should be registered to the registry. By autonomy, components are totally blind to any other component technologies. If a component is aware of its collaborators, it is expecting its collaborators are of the same technology as itself. Each component offers some services that are identifiable in terms of business domain features.

Thus, the interoperation means the communication across the territory boundaries. There are two main tasks in this communication: first, where is the component; second, how do components communicate. URDS [Sir02] takes care of the first task by searching federated registries in the business domain for expected components and returning with the registry and the component ID. This paper specifically addresses the second task. The interoperation is achieved by generating proxies dynamically for invoking the components from the registry and for replaying communications. Shown in the figure 2, the communication between the component and the proxy falls in the same territory. The essential aspect of interoperation in this picture is to establish a common message protocol so that proxies can talk to each other across the technology boundaries. In UniFrame, we use Simple Object Access Protocol (SOAP)[3] for encoding and decoding parameters, data types and exceptions. The code that actually realizes the interoperation is called the glue/wrapper code, which includes two proxies.

[3] SOAP Messaging Framework, W3C, http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

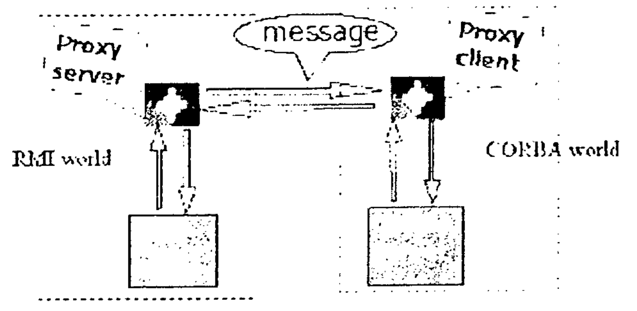


Figure 2. The Interoperation Framework

To be specific, the fundamentals of the UniFrame interoperation framework are as follows:

1. The glue/wrapper establishes a binary connection for any two heterogeneous components. Between these two components, one must be the service requester, and the other one must be the service provider. From this perspective, no matter what is the underlying architecture of the whole distributed system, client-server is a general framework for a binary relationship of a pair of communicating components. For the communication between the two components we need a proxy server for the service requester, and a proxy client for the service provider. The proxy server registers itself to its component registry listening for the request coming from the service requester, and then translates this request through the SOAP channel to the proxy client who decodes the SOAP message and invokes the ultimate service provider with the redirected service request. Two proxies also take the responsibility of managing the communication session. The use of proxies attacks the problem of the heterogeneity of component models; and SOAP/HTTP solves the language heterogeneity and distribution.
2. The glue/wrapper code realizes the interoperation at run time, i.e., the existing component should not be modified or recompiled. The glue/wrapper can be generated, compiled and bound dynamically during the composition run time.
3. Because the semantics of business domain features are standardized and shared by all the feature implementation developers, each implementation can have slightly customized interfaces including different naming strategies of parameters and methods, and the variations on the parameters (only to a degree that the translations can be done automatically for solving the variations).

The main challenge in realizing interoperation among heterogeneous components is not the issue of constructing glue/wrapper code for a particular pair of components, but to construct a generator that can automatically generate glue/wrapper code for different pairs of components on demand. To achieve that, the generator needs to access both the knowledge for the technology domains at the domain level and the knowledge for a particular component implementation at the component level. For the technology domain, the generator has to know how many kinds of technology domains (component model domains, programming language domains, operating system platform domains, etc.) and what information in a particular technology domain (e.g. Java programming language domain) for the interoperation purpose. At the component level, the generator needs to know from the deployment aspect of UMM what technologies are employed in a component implementation.

In the next section, we will review an informal implementation of the glue/wrapper code generator.

3.2. An Informal Implementation of Glue/Wrapper Code Generator

In this informal implementation, both the generator and the technology domain knowledge are written in Java. Domain knowledge is embedded in the java classes in the form of printing statements. Shown in figure 3, there are 4 different kinds of technology domains that the generator directly accesses: proxy client, proxy server, programming languages, and operating systems. The proxy server and the proxy client inherit the architecture knowledge from the architecture domain server and client respectively. There can be federated hierarchies in each technology domain. For example, for a specific component model, say Remote Method Invocation (RMI)[4], there is an RMIServer that implements ProxyServer and extends RMI, also there is an RMIClient (although not shown in figure 3) that implements ProxyClient and extends RMI. Then we will be able to generate both proxy server and proxy client for a RMI component. A component model is usually abstract and should be concretized by different vendor-specific technologies. For example, TAO [Har98] is a concretization of CORBA, and JavaRMI is a concretization of RMI.

There are some benefits in developing the generator in Java.

1. By taking advantages of polymorphism, the generator is generic to any specific technology as it only deals with interfaces.
2. By using Java reflection, we can dynamically load a specific technology domain class as needed based on

[4] Java Remote Method Invocation (Java RMI), http://java.sun.com/products/jdk/rmi/

the parameters in the component UMM. For example, if the UMM indicates the language used for two components are Java and C++, then only the Java and C++ classes in programming language domains are loaded into the Java runtime environment. This will drastically improve the performance of the generator considering technology domains contain a wide variety of classes.

3. The generator framework is extensible. We can extend the framework with any programming languages, operating systems and component models. In the case of new technologies (a new component model, a new vendor-specific product for an existing component model, etc.), we only need to modify the framework by adding the new technology domain subclass, and the generator should remain unchanged.

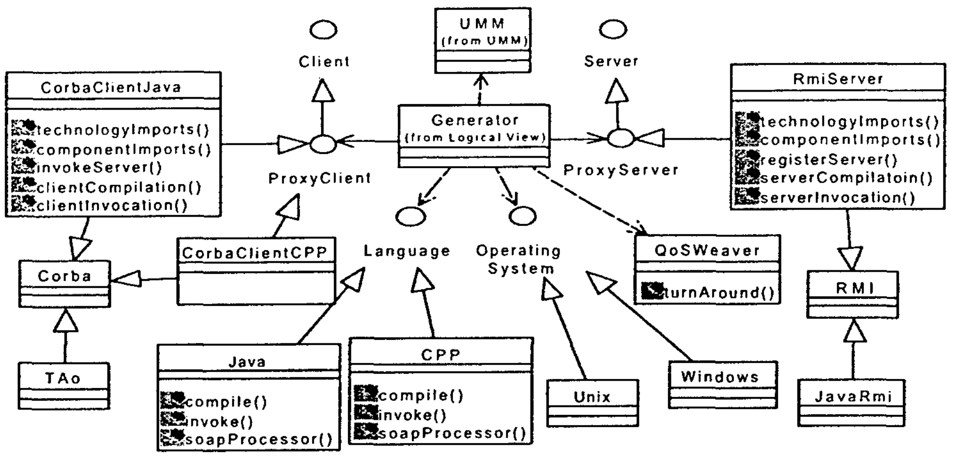


Figure. 3 The Glue/Wrapper Code Generator in Java

By constructing a technology domain knowledge base, we do not mean constructing a complete specification for a particular technology. For the interoperation under the hypothesis mentioned earlier, only some specific information is needed. Such information includes: how to register and invoke a server from a registry in a specific component model; how to process SOAP messages in a specific language; how to compile and invoke a program in a specific programming language and operating system platform; what are the component model product specific class path and compilation options.

Besides generating interoperation code, the generator has other responsibilities such as dynamic QoS testing, system monitoring, and session management probe generation. As an example, we can use AspectJ [Kic97] to weave turnaround time testing code into the generated proxies (shown in figure 4 as QoSWeaver class).

3.3. Towards the Formalization of Automated Glue/Wrapper Code Generation

In the previous section, we have sketched out some benefits of implementation in Java. However, embedding the technology domain knowledge into a programming language using printing statements tends to blur the technology domain specific information. Consequently, it will be an obstacle for domain evolution and reuse, and further prevent the generator from evolving.

To solve this problem, we have applied the Generative Programming (GP) [Cza00] and Product-line Architecture [Cle01], [SEI02], [Wei99]. Both of these technologies aim at defining and modeling a family of products so that a product instance can be generated automatically from this family. As mentioned in section 2.1, the system family development is the core design of UniFrame, and as well as the interoperation framework in UniFrame. The rationale for the applicability of GP is that the glue/wrapper code for a pair of components of particular technologies is one product instance; the glue/wrapper code for the pairs of components of all possible technologies form a family of glue/wrapper code. If this family can be well modeled, one particular glue/wrapper code instance can be generated from the family automatically.

The GDM for the family of glue/wrapper code is called the Interoperation GDM (IGDM, see figure 4). IGDM straddles different technology domains including different component model domains, different programming language domains, different operating system domains, and different security method domains. The feature model in the IGDM explicitly models the domain-specific features of different technology domains, which direct the variations among glue/wrapper code instances. The generation of glue/wrapper code for components in different technologies depends on the domain-specific features of technology domains. In the IGDM, the reusable components, from which the glue/wrapper code can be generated, are the code fragments of potential glue/wrapper code.

In order to support the automated glue/wrapper code generation from the IGDM, we have adopted a formal modeling theory on feature modeling in the IGDM. The feature model in the IGDM is defined as a language; the glue/wrapper code generated from the IGDM is a valid sentence of this language. The generator for the glue/wrapper code is the interpreter for the grammar that is used to define the feature model. The terminal symbols of the grammar are code fragments. The glue/wrapper code is a string of code fragments.

To apply successfully this theory and the programming-language-oriented techniques to feature modeling, the first question to be answered is whether there exist concepts in feature models that are the counterparts of syntax and semantics in programming languages. The fact is these concepts do exist in the feature models, and are discussed below.

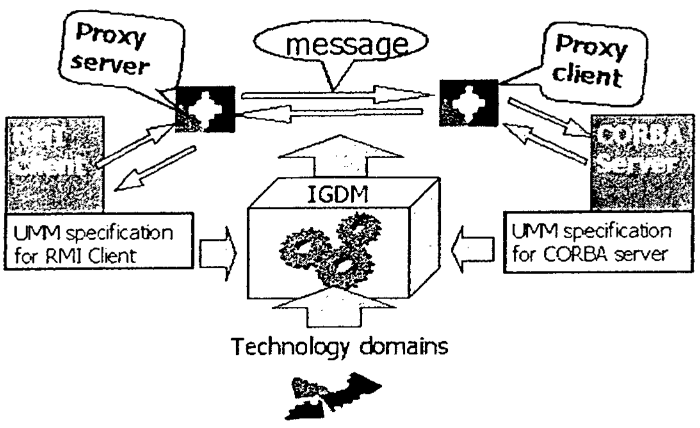

Proxy server
message
Proxy client
IGDM
UMM specification for RMI Client
UMM specification for CORBA server
Technology domains

Figure. 4 The Formalization of Automated Glue/Wrapper Code Generation

1. The composition syntax is the structure of the interoperable framework. The following context-free derivations show part of the structure of the glue/wrapper code to be generated. Currently, the grammar we use to define IGDM feature model is called TLG++ [Zha04]. The following code is in TLG++. For the notational syntax, the "," is for "and", and the ";" is for "or".

```
glueWrapper code : proxyServer,
  proxyClient.
proxyClient : technologyImports,
  componentImports, invokeServer,
  clientCompilation, clientInvocation.
invokeServer : findRegistry,
  getServerObject, initiateServer,
  serverInvocationExceptions.
......
```

2. Static semantics constrains types of glue/wrapper code to be generated. In particular, the component model is modeled as the type of the component; and programming languages, operating systems, message's signature and type, security methods, and digital signatures are modeled as the attributes of the components. Based on the different value of component type and its parameters, different glue/wrapper should be generated. In the following code fragments, the codes in bold are the parameters that indicate the different features of technology domains. TLG++ distinguishes itself from context-fee grammars is this feature of parameterization. The parameters are evaluated while the syntax tree is built. The codes underlined are the glue/wrapper code fragments enclosed in the double quotation mark. The code fragments are the terminals of the grammar.

```
findRegistry:
  where ComponentModel= corba,
  "orb=  org.omg.CORBA.ORB  .init(args,
null);"
   ProductTraderPackage
  "trading=TradingHelper.narrow
```

```
 (orb.resolve initial references("LCBT
rading" ));"......;
where ComponentModel = rmi,
    ......;
where ComponentModel = j2ee......
```

3. Dynamic semantics models the component composition QoS that are affected by the component technologies. If the components are implemented in different technologies, they will present different QoS values. The generated glue/wrapper code will also affect the QoS, and should be part of dynamic semantics. Event grammars [Aug97] are used to generate an event trace, which acts as the QoS metric to be inserted into the generated glue/wrapper code.

4. Related Work

There have been some attempts towards achieving interoperability among different technologies emerging out of industry and research organizations. Some prominent examples, besides the work mentioned in section 3.1, are described below.

Middleware technologies such as CORBA [Corba] and DCOM [Ses97] provide a communication infrastructure for a heterogeneous and distributed collection of objects. Based on this infrastructure, objects can interoperate across networks regardless of the language in which they are written or the platform on which they are deployed. However such middleware or component models exclude the presence of others. UniFrame gives a vision of unified middleware providing the interoperation not only among the programming languages and platforms but also among the component models. The proxies in this paper are similar to the stubs/skeletons in CORBA. However, the concept of IDL in CORBA is elevated to the business feature model of this paper. The feature model in a business domain defines the semantics of features and their interactions, and is shared by the feature implementation developers.

Some ad hoc approaches for interoperation between component models come out from the industry that are targeting specific component model pairs. RMI is a language centric approach using JRMP (Java Remote Method Protocol) for interactions between distributed objects. RMI requires that the entire distributed application be programmed in pure Java. Sun[5] and IBM[6] have jointly developed RMI-IIOP, a new version of RMI that runs over IIOP and interoperates with CORBA ORBs and CORBA objects programmed in other languages. To bridge CORBA and DCOM, the Object Management Group (OMG) provides the interworking architecture specifications regarding the mappings between DCOM and CORBA which includes: Interface Mapping, Interface Composition Mapping and Identity Mapping, etc. [Rap01].

Web services [New02] claims to be a means of interoperation among component models. Nevertheless, web services achieve the interoperation by introducing yet more standards such as Web Service Definition Language (WSDL), Universal Description, Discovery, and Integration (UDDI), and SOAP. This does not completely solve the problem due to the inherited local autonomy and the difficulty of the adoption of standards, whereas UniFrame approaches the problem in a different way by modeling existing technology domains.

As mentioned in section 3.1, MDA [Fra03] has subscribed to the meta-interoperation approach. For example, for the interoperation between the web service and Java, the system has to know the following three things: the platform-independent UML class model, the UML-java mapping, the UML-SOAP/WSDL mapping. As with web services, MDA forces UML or MOF to be the standards for the interoperation.

5. Conclusions

In this paper, we have discussed an interoperation framework for integration of heterogeneous and distributed software components. The target goal of this framework is the automated glue/wrapper code generation during the component assembly time. This framework incorporates the following key concepts: 1) an introspective meta model (UMM) for the autonomous components; 2) an explicit modeling of domain knowledge of various technology domains instead of introducing new standards for interoperation; 3) introducing the IGDM that models a family of glue/wrapper code to provide a formal foundation for automated glue/wrapper code generation; 4) a language-oriented way to formalize the IGDM so that the glue/wrapper code generated from IGDM is a valid sentence that can be generated from a grammar. The initial experiments have been carried out to integrate components written in RMI and CORBA, and the glue/wrapper code can be automatically generated for their

[5] Sun Microsystems, Java RMI-IIOP Documentation url: *http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/index.html*

[6] IBM developer Works, Java technology Standards RMI-IIOP, url: *http://www-106.ibm.com/develperworks/java/rmi-iiop/summary.html*

interoperation based on the informal implementation approach. Future work will be to design and extend our grammar notation to formalize IGDM. Experiments are also done on applying this framework to other component models such as .Net, DCOM, J2EE, Web Services, mobile agents, and as well as wireless component models [Sha03].

6. Acknowledgement

This research is supported in part by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

7. References

[Aug97] M. Auguston, A. Gates, M. Lujan, "Defining a Program Behavior Model for Dynamic Analyzers," *Proc. SEKE '97, 9th Int. Conf. Software Eng. Knowledge Eng.*, pp. 257-262, 1997.

[Bax04] I. Baxter, C. Pidgeon, M. Mehlich, "DMS: Program Transformations for Practical Scalable Software Evolution", to appear in the Proc. of 2004 International Conference on Software Engineering (ICSE), 2004.

[Ben87] S. Bendifallah and W. Scacci, "Understanding Software Maintenance Work", *IEEE Transactions on Software Engineering*, Vol. 13 , No. 3, 1987.

[Bra02] G. J. Brahnmath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, C. C. Burt, "A Quality of Service Catalog for Software Components," *Proc. Southeastern Software Engineering Conf.*, pp. 513-521, 2002.

[Bry03] B. Bryant, B-S. Lee, F. Cao, W. Zhao, C. Burt, J. Gray, R. Raje, A. Olson, M. Auguston, "From Natural Language Requirements to Executable Models of Software Components", *Proc. of* the *Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*, pp. 51-58, 2003.

[Cao03] F. Cao, Z. Huang, B. Bryant, C. Burt, R. Raje, A. Olson, M. Auguston. ``Automating Feature-Oriented Domain Analysis," Proc. of the 2003 International Conference on Software Engineering Research and Practice (SERP'03), CSREA Press, pp. 944-949, 2003.

[Cle01] P. Clements, L. Northrop, *Software Product Lines: Practice and Patterns,* Addison-Wesley, 2001.

[Corba] Common Object Request Broker Architecture (CORBA), http://www.corba.org/

[Cza00] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Gro02] T. Grose, G. Doney, S. Brodsky, *Mastering XMI*, John Wiley & Sons, Inc., 2002.

[Fra03] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley Publishing, Inc., 2003.

[GME] GME User's Manual. The Institute for Software Integrated Systems, Vanderbilt University. http://www.isis.vanderbilt.edu/Projects/gme/Doc.html

[Har98] T. Harrison, D. Levine, D. Schmidt, "The Design and Performance of a Real-time CORBA Event Service", *Computer Communications*, Vol. 21, No. 4, 1998

[Kan90] K. C. Kang, S, G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, 1990.

[Kan98] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Engineering* 5, pp. 143-168, 1998.

[Kic97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, J. Irwin, "Aspect-Oriented Programming", *Proc. of European Conference for Object-Oriented Programming (ECOOP)*, pp. 220-242, Springer-Verlag, 1997.

[Lee02a] B.-S. Lee, B. R. Bryant, "Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language", *Proc. of ACM Symposium on Applied Computing (SAC)*, pp. 932-936, 2002.

[Lee02b] Lee, B.-S. and Bryant, B. R., "Automation of Software System Development Using Natural Language Processing and Two-Level Grammar," *Proc. 2002 Monterey Workshop Radical Innovations of Software and Systems Engineering in the Future*, 2002, pp. 244-257.

[New02] E. Newcomer, *Understanding Web Services: XML, WSDL, SOAP, and UDDI,* Addison-Wesley, 2002.

[Raj00] R. R. Raje, "UMM: Unified Meta-object Model for Open Distributed Systems." *Proc. ICA3PP 2000, 4th IEEE Int. Conf. Algorithms and Architecture for Parallel Processing*, 2000, pp. 454-465.

[Raj01] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C Burt, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," *Proc. Monterey Workshop Engineering Automation for Software Intensive System Integration*, pp. 109-119, 2001.

[Raj02] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt, "A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software

Components," *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 12, pp. 1009-1034, 2002.

[Rap01] Raptis, K., Spinellis, D., Katsikas, S. "Multi-Technology Distributed Objects and their Integration," *Computer Standards & Interfaces, Vol. 23, 157-168, 2001.*

[Sha03] P. V. Shah, B. R. Bryant, C. C. Burt, R. R. Raje, A. M. Olson, M. Auguston, "Interoperability between Mobile Distributed Components using the UniFrame Approach," Proc. of the 41st Annual ACM Southeast Conference, pp. 30-35, 2003.

[SEI02] Software Engineering Institute, A framework for software product line practice –version 3.0, 2002, http://www.sei.cmu.edu/plp/framework.html

[Ses97] R. Sessions, *COM and DCOM: Microsoft's Vision for Distributed Objects*, New York, NY: John Wiley & Sons, 1997.

[Sir02] N. N. Siram, R. R. Raje, B. R. Bryant, A. M. Olson, M. Auguston, C. C. Burt, "An Architecture for the UniFrame Resource Discovery Service," *Proc. SEM 2002, 3rd Int. Workshop Software Engineering and Middleware*, Springer-Verlag LNCS, Vol. 2596, pp. 20-35, 2002.

[Szy02] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd edition, Addison-Wesley Longman, 2002.

[Vin97] S. Vinoski, "CORBA: Integration Diverse Applications Within Distributed Heterogeneous Environments", *IEEE Communications*, Vol. 14, No. 2, 1997.

[Wei99] D. M. Weiss, C. T. R. Lei, *Software Product-line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[Zha02] W. Zhao, B. R. Bryant, F. Cao, R. R. Raje, M. Auguston, A. M. Olson, C. C. Burt. "A Component Assembly Architecture with Two-Level Grammar Infrastructure", *Proc. of OOPSLA'2002 Workshop Generative Techniques in the Context of Model Driven Architecture*, 2002. http://www.softmetaware.com/oopsla2002/zhaow.pdf

[Zha04] W. Zhao, B. R. Bryant, R. R. Raje, M. Auguston, C. C. Burt, A. M. Olson, "Grammatically Interpreting Feature Compositions", to appear in the proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE'04), 2004.

Formal Specification of Generative Component Assembly Using Two-Level Grammar *

Barrett R. Bryant
Carol C. Burt
Computer/Information Sci.
Univ. Alabama-Birmingham
Birmingham, AL 35294, USA
bryant@cis.uab.edu
cburt@cis.uab.edu

Mikhail Auguston
Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
mikau@cs.nmsu.edu

Rajeev R. Raje
Andrew M. Olson
Computer/Information Sci.
Indiana Univ. Purdue Univ.
Indianapolis, IN 46202, USA
rraje@cs.iupui.edu
aolson@cs.iupui.edu

ABSTRACT

Two-Level Grammar (TLG) is proposed as a formal specification language for generative assembly of components. Both generative domain models and generative rules may be expressed in TLG and these specifications may be automatically translated into an implementation which realizes an integration of components according to the principles of the Unified Meta-component Model (UMM) and Unified Approach (UA) to component integration. Furthermore, this implementation realizes Quality of Service (QoS) guarantees by means of static QoS verification at the time of system assembly, and dynamic QoS validation on a set of test cases.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages, tools*; D.2.11 [**Software Engineering**]: Software Architectures—*languages*; D.2.12 [**Software Engineering**]: Interoperability—*distributed objects*

General Terms

Languages

Keywords

Component-based software, formal specification, generative programming, Two-Level Grammar

*This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant numbers DAAD19-00-1-0350 and 40473-MA, and by the U. S. Office of Naval Research under award number N00014-01-1-0746.

1. INTRODUCTION

The recent shift in the focus of OMG (Object Management Group) to "Model Driven Architecture" (MDA) [10] is a recognition that to create mechanized software and bridging of component architectures requires standardization not only of infrastructure but also Business and Component Meta-Models. This emphasizes the fact that a comprehensive meta-model, that seamlessly encompasses heterogeneous components by capturing their necessary aspects including Quality of Service (QoS) and associated guarantees, is needed for creating future generation of distributed systems. The UniFrame project proposes a Unified Meta-component Model (UMM) [11] for distributed component-based systems, and a Unified Approach (UA) [11] for integrating these components. Component development and deployment starts with a UMM requirements specification of a component from a particular domain. This specification is natural language-like and indicates the functional (i.e., computational) and non-functional (i.e., QoS parameters) features of the component. This specification is then refined into a formal specification, based upon the theory of Two-Level Grammar (TLG) [5]. Generative domain models and generative rules for system assembly [6] may be expressed in TLG and these specifications may be automatically translated into an implementation which realizes an integration of components.

2. THE UNIFIED APPROACH

The distinctive features of the Unified Approach are:

- The developer of the desired distributed system presents to this process a system query, in a structured form of natural language, that describes the required characteristics of the distributed system. The query is processed using the domain knowledge (such as key concepts from a domain) and a knowledge-base containing the UMM description of the components for that domain. From this query a set of search parameters is generated which guides "head-hunter" agents for a component search in the distributed environment. Head-hunters serve to locate the components which are needed to complete the requested system [12].
- A set of potential components is collected for that do-

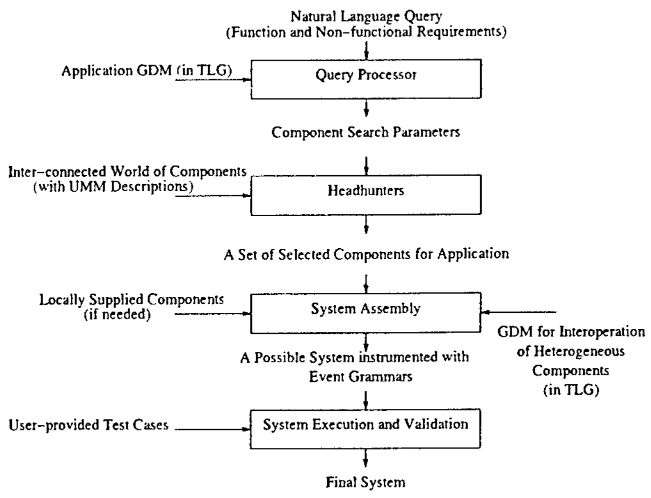

Natural Language Query
(Function and Non-functional Requirements)
Application GDM (in TLG)
Query Processor
Component Search Parameters
Inter-connected World of Components
(with UMM Descriptions)
Headhunters
A Set of Selected Components for Application
Locally Supplied Components
(if needed)
System Assembly
GDM for Interoperation
of Heterogeneous
Components
(in TLG)
A Possible System instrumented with
Event Grammars
User-provided Test Cases
System Execution and Validation
Final System

Figure 1: System Assembly in UniFrame

main, each of which meets the Quality of Service (QoS) requirements specified by the developer. QoS requirements are expressed in terms of a catalog of parameters established for this purpose [4]. After the components are fetched, the system is assembled according to the generation rules embedded in the generative domain model. Essentially, the generated code constitutes the glue/wrapper interface between the components.

- Along with the generated system will be a formal UMM specification of the generated system so that it may be used in subsequent assemblies. This formal UMM specification will also be a basis for generating a set of test cases to determine whether or not an assembly satisfies the desired QoS.

- Static QoS parameters (e.g. dependability of the component) are processed during generation time. Dynamic QoS parameters (e.g. response time of the component) result in instrumentation of generated target code based on event grammars [1, 2], which at run time produce the corresponding QoS dynamic metrics which may be measured and validated.

QoS parameters require instrumentation necessary for the run-time QoS metrics evaluation. Based on the query or informal requirements, the user has to come up with a representative set of test cases. Next the implementation is tested using the set of test cases to verify that it meets the desired QoS criteria. If it does not, it is discarded. After that, another implementation is chosen from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is found, or until the collection is exhausted. In the latter case, the process may request additional components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. If a satisfactory implementation is found, it is ready for deployment. The complete view of this system is shown in Figure 1.

A few attempts have been made to incorporate QoS into component-based software systems. The Aster project [7] uses architectural descriptions of components and their interactions, including non-functional properties, to customize middleware. Quality Objects (QuO) [3], a framework for providing QoS to software applications composed of objects distributed over wide area networks, bridges the gap between socket-level QoS and distributed object level QoS, emphasizing specification, measuring, controlling, and adapting to changes in QoS. RAPIDware [8], an approach to component-based development of adaptable and dependable middleware, uses rigorous software development methods to support interactive applications executed across heterogeneous networked environments. Process^{NFL} [9] is a language for describing non-functional properties of software, which may include QoS properties. The Unified Approach is concerned not only with specifying QoS properties of components, but also to assure satisfaction of these properties in an implementation resulting from assembling the components. It should be noted that the assurance of QoS (as described above) indicates that a component can guarantee appropriate values for its QoS parameters in an 'ideal' situation. This does not guarantee that a component will be able to either provide this QoS under failure circumstances or will automatically adjust its QoS to hide the failures. For the failure situations, the ideas provided by Aster, QuO, or RAPIDware can be incorporated.

3. TWO-LEVEL GRAMMAR SPECIFICATION

Two-Level Grammar (TLG) is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The "two levels" are two context-free grammars defining the set of type domains and the set of function definitions operating on those domains, respectively. These grammars may be defined in the context of a class in which case type domains define instance variables of the class and function definitions define methods of the class. The TLG formalism is used to specify the generative rules needed for component assembly and the output of the TLG will provide the desired target code (e.g., glue and wrappers for components and necessary infrastructure for distributed run-time architecture). All of this is implemented according to the process for translating TLG specifications into executable code [5].

We illustrate the formal specification of generative rules using TLG by means of a simple bank account management system. The specification of a bank account should include its attributes and the operations it should perform, such as `check balance`, `deposit`, or `withdraw`. Assume that the GDM in this example contains a rule for system assembly that specifies that a Bank Account Management System consists of one of each of the two component types, *AccountServer* and *AccountClient*, each of which follows the bank account feature model. Further, let there be two instances of *AccountServer* and one instance of *AccountClient*. Server components are heterogeneous – `JavaAccountServer` adheres to the Java-RMI model and has methods `javaDeposit`, `javaWithdraw`, and `javaBalance`, and QoS parameters *Availability* $\geq$ 85% and *Response Delay* $<$ 30 *ms*; while `CorbaAccountServer` uses the CORBA model and has methods `corbaDeposit`, `corbaWithdraw`, and `corbaBalance`, and QoS parameters *Availability* $\geq$ 90% and *Response Delay* $<$ 10 *ms*. The client, `JavaAccountClient`, is developed by using the Java-RMI model, with calls to server `depositMoney`, `withdrawMoney`, and `checkBalance`, and QoS parameters *Availability* $\geq$ 90% and *Response Delay* $<$ 50 *ms*. The goal is to assemble a bank account management system from these available components.

Queries are stated in a structured form of natural language. The general form of a query is to request creation of a system that has certain QoS parameters. The name of the system is important in identifying the application domain and the QoS parameters should also follow the catalog standards. A sample query for the above example can be informally stated as: *Create a bank account management system that has availability $\geq$ 50% and response delay $<$ 100 ms.* This query requires the satisfaction of one static and one dynamic QoS parameter. From the query and the available knowledge in the GDM associated with bank account management systems, a query will be formulated for a headhunter in the UMM. In response, the headhunter will discover the three components and their QoS properties. Note that the availability QoS parameter is used to screen potential components at the time they are retrieved. The catalog specification for this parameter suggests that the availability criteria should be multiplied, so the availability of the Java-Java system is 76.5% and for the Java-CORBA system 81%, both meeting the stated criteria.

TLG is used as the formalism for both the UMM and generative rules. The UMM formalization establishes the context for which the generative rules may be applied. TLG functions include generative rules for construction of wrapper/glue code and event grammar instrumentation to assure the QoS of the bank account record management system. The GDM for bank account management systems will be described according to this template, including both generation rules and QoS parameter processing.

A sampling of TLG rules which may be used to generate the appropriate glue/wrapper code to connect the components of the bank account management system is presented below. These rules are based on selecting from the GDM for bank account management systems the appropriate system model for this two-component DCS. The generation rule to produce Java code for two UMM models representing a client and server, respectively, is expressed using a TLG function which has a signature followed by a set of subfunctions to be executed when the main function is called. Function keywords are indicated in bold while class/object names are italicized.

```
generate system from ClientUMM and ServerUMM :
    ClientOperations := ClientUMM get operations,
    ServerOperations := ServerUMM get operations,
    OperationMapping :=
        map ClientOperations into ServerOperations,
    ComponentModel :=
        ServerUMM get component model,
    generate java code for OperationMapping
        using ComponentModel.
```

The main tasks are to map client operations onto server operations, e.g., `depositMoney` in `JavaAccountClient` maps to `corbaDeposit` in `CorbaAccountServer` or to `javaDeposit` in `JavaAccountServer`, and then generate the code to implement this mapping. The next set of rules describes the specifics of generating CORBA code in Java to implement the mapping that arises by integrating the `JavaAccountClient` with the `CorbaAccountServer`, including the mechanism for generating individual methods. The generated code is distinguished from types (variables) and function keywords by using a typewriter font.

```
generate java code OperationMapping using corba :
    CorbaPackageName :=
        OperationMapping get corba package name,
    CorbaObjectClass :=
        OperationMapping get corba object type,
    ClassName := OperationMapping get class name,
    JavaClassName := Java || ClassName,
    CorbaObjectName := object || ClassName,
    SetUpCode := ComponentModel generate java code,
    Operations :=
        generate java code for OperationMapping,
    return
        import CorbaPackageName .  *;
        public class JavaClassName {
            private CorbaObjectClass CorbaObjectName ;
            // initialize CORBA client module
            public void init () {
                SetUpCode
            }
            Operations
        }.
```

The class structure required by the Java implementation consists of a function `init` to set up the CORBA ORB and the operations needed in the server. This includes the code to initialize the CORBA object so that future operations can refer to it. It is necessary to first extract the names of the CORBA package, class of the CORBA object to be referenced within the package, and the name of the class itself. These are all stored in the *OperationMapping*. The name of the Java class generated is simply the string "Java" concatenated [1] with the name of the server class, i.e., `JavaCorbaAccountServer`. The name of the CORBA object is generated in a similar way. For simplicity, only the case where the class is to contain a single method is shown. Multiple methods are handled similarly.

```
generate java code for
        OperationName1 ArgumentList1 ReturnType
        maps to
        OperationName2 ArgumentList2 ReturnType :
    JavaReturnType := java type of ReturnType,
    JavaArgumentList :=
        list all Argument from ArgumentList1
            mapped to JavaArgument
                by function java argument of
                    Argument is JavaArgument,
    JavaArgumentListDefinition :=
        separate JavaArgumentList by , ,
    OperationCall := generate java code for
        OperationName2 ArgumentList1 ReturnType,
    return
        public JavaReturnType OperationName1
              ( JavaArgumentListDefinition ) {
            EventTrace .  setBeginTime ();
            OperationCall
            EventTrace .  setEndTime ();
            EventTrace .  calculateResponseTime ();
        }.
```

[1]The TLG concatenation operation (||) differs from juxtaposition in that it does not produce a space between the operands.

This generation assumes that the methods have the same return type and so the main task is to express the arguments of the first operation in terms of Java syntax, generate the appropriate method call, and instrument the code with the event grammar mechanism to measure the response time. The former is accomplished by using a TLG list comprehension to map the arguments in *ArgumentList1* into corresponding Java arguments represented by *JavaArgumentList*. Each *Argument* from *ArgumentList1* is mapped into a *JavaArgument* using the function **java argument of** *Argument* **is** *JavaArgument*. There is a subtlety here in that *JavaArgumentList* is an abstract syntax representation of the desired argument list and so this must be made into concrete syntax using the **separate** operation which adds the appropriate commas in between the argument declarations. The appropriate method call is handled by the rule below.

```
generate java code for
      OperationName ArgumentList ReturnType :
  IdList := list all Argument from ArgumentList
      mapped to Id by
          function argument id of Argument is Id,
  IdListInCall := separate IdList by , ,
  return CorbaObjectName . OperationName
      ( IdListInCall ) ;.
```

Again a list comprehension is used to extract the arguments from the argument list, this time only the identifier part (achieved by **function argument id of** *Argument* **is** *Id*). Likewise, the abstract syntax representation must be made concrete by comma separators.

Finally, the event grammar instrumentation is added to measure the time at the beginning of the server method call and again at the end so that the actual response time can be evaluated against the required QoS ($< 100ms$). The QoS metrics for "response delay" mean execution time for each method call within the server or client, and require the instrumentation of each generated wrapper for the client/server method call with auxiliary functions able to check the clock at the beginning and at the end of method call, calculate the duration, and submit it to the execution monitor (also generated as a part of instrumentation). We assume that these are taken care of by a class called `EventTrace`. Each of the two example systems will be implemented with the code for carrying out event trace computations according to test cases which must be supplied by the user. These test cases will be executed to verify that the bank account management system satisfies the QoS specified in the query. If the system is not verified, it is discarded. This verification process is carried out for each of the generated bank account management system (two in the above example). Then the one with the best QoS is chosen, in the above example the `CorbaAccountServer` and `JavaAccountClient` combination.

For the example, the following code for the `depositMoney` function would be produced.

```
public void depositMoney (float ip) {
  EventTrace . setBeginTime ();
  objectCorbaAccountServer . deposit (ip);
  EventTrace . setEndTime ();
  EventTrace . calculateResponseTime ();
}
```

In the future, the efficient generation and update of a distributed computing system will require at least a semi-automatic integration of software components, based on their advertised QoS, in such a way that it meets the QoS constraints specified by the user. UniFrame facilitates semi-automatic construction of such a system. A simple case study is provided in this paper for illustration, but the principles of the proposed approach can be applied to larger applications.

4. REFERENCES

[1] M. Auguston. Program behavior model based on Event Grammar and its application for debugging automation. In *Proc. 2nd Int. Workshop Automated and Algorithmic Debugging*, pages 277–291, 1995.

[2] M. Auguston. Tools for program dynamic analysis, testing, and debugging based on event grammars. In *Proc. SEKE 2000, 12th Int. Conf. Software Engineering Knowledge Engineering*, pages 159–166, 2000.

[3] BBN Corporation. *Quality Objects (Quo)*, *http://www.dist-systems.bbn.com/tech/QuO*, 2001.

[4] G. J. Brahnmath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, and C. C. Burt. A quality of service catalog for software components. In *Proc. $(SE)^2$ 2002, Southeastern Software Engineering Conf. (to appear)*, 2002.

[5] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an object-oriented requirements specification language. In *Proc. 35th Hawaii Int. Conf. System Sciences*, 2002.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[7] INRIA-Rocquencourt. *ASTER: Software Architectures for Distributed Systems, http://www-rocq.inria.fr/solidor/work/aster.html*, 2001.

[8] Michigan State University. *RAPIDware: Component-Based Development of Adaptable and Dependable Middleware, http://www.cse.msu.edu/rapidware*, 2001.

[9] N. S. Rosa and P. R. F. Cunha and G. R. R. Justo. process^{nfl}: A language for describing non-functional properties. In *Proc. 35th Hawaii Int. Conf. System Sciences*, 2002.

[10] Object Management Group (OMG). Model Driven Architecture: A technical perspective. Technical report, OMG Document No. ab/2001-02-01/04, February 2001.

[11] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, and C. C. Burt. A unified approach for the integration of distributed heterogeneous software components. In *Proc. Monterey Workshop Engineering Automation for Software Intensive Systems*, pages 109–119, 2001.

[12] N. N. Siram, R. R. Raje, B. R. Bryant, A. M. Olson, M. Auguston, and C. C. Burt. An architecture for the UniFrame Resource Discovery Service. In *Proc. SEM 2002, 3rd Int. Workshop Software Engineering Middleware (to appear)*, 2002.

Analyzing the Web Services and UniFrame Paradigms[1]

Natasha Gupta[2] Rajeev R. Raje[2] Andrew Olson[2] Barrett R. Bryant[3] Mikhail Auguston[4] Carol Burt[3]

Abstract

The software realization of today's distributed systems often require combining of heterogeneous software components, each offering a specialized service. This heterogeneity necessitates a paradigm for the interoperation of different components. Various models and approaches have been proposed to facilitate a smooth interoperation. Web Services and UniFrame are two such paradigms. This paper presents analyses of these two alternatives, thereby, indicating their similarities and differences.

1 Introduction

The evolution in the field of computing has shifted its paradigm from a centralized one to a distributed one. Hence, the target environment is no more a centrally managed, but concerned with collaboration, data sharing, and other new modes of interactions involving distributed resources. This necessitates the availability of technologies and solutions that can effectively and efficiently integrate services across disparate systems. This integration can be challenging because of the need to achieve various qualities of services when running on top of different native platforms [1]. Innovations in this field have led to developments of many paradigms including Web Services (WS's) [2], and UniFrame [3]. Each of these approaches has associated pros and cons. Web Services have emerged as a new "Web Development Tool" which enables a web application to become more interactive, by providing means to make it communicate at the middle-tier lever (business logic level) and provide a new platform to build software for a distributed environment. UniFrame is a research project that aims to provide a framework that allows a seamless interoperation of heterogeneous components. The purpose of this paper is to compare and contrast the Web Services framework and the UniFrame.

2 Related Work

2.1.1 Enterprise Application Integration (EAI) Solutions

The EAI [4] solutions provide the infrastructures for an organization that take the integration technology from the traditional point-to-point connections to a level that links multiple applications and databases internal to the organization to share information and business processes. EAI typically uses middleware to connect to different applications. A custom interface is built to link each separate application in the EAI system. Most EAI systems use adaptors to connect applications. Several types of EAI exist, including data integration, business process integration and method integration. However,.the integration that EAI solutions provide tends to be complex and expensive, despite improving the overall communication. In addition, the EAI interfaces are not reusable and cannot be used by a company to connect to their business partners whose applications fall outside the boundaries of the organization. Web Services overcome this limitation by providing a set of reusable interfaces to applications, which enables them to interoperate with any other application (Web Service) using SOAP.

2.1.2 Business-To-Business (B2B) Solutions

The Internet has given birth to a "digital economy" [5]. In such an economy, B2B e-commerce provides a company with an effective and efficient end-to-end process communication to buy and sell services in an economical way. B2B relationships are often characterized by stringent requirements for security, auditability, availability, service level agreements and complex transaction processing flows [1] in addition to the large technical differences that arise between different organizations. B2B Integration has long been accomplished with the use of technologies like Enterprise Data Interchange (EDI). EDI is a relatively arcane technology that requires substantial overhead on the

[1] This material is based upon the work supported by the U.S. Office of Naval Research under award number N0014-01-1-0746.

[2] Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 723 W. Michigan St., SL280, Indianapolis, IN 46202, USA, {nsgupta, rraje, aolson}@cs.iupui.edu.

[3] Department of Computer and Information Sciences, University of Alabama at Birmingham, 1300 University Blvd., CH 115A, Birmingham, AL 35294, USA, {bryant, cburt}@cis.uab.edu.

[4] Computer Science Department, Naval Postgraduate School, 833 Dyer Rd., SP517, Monterey, CA 93943, USA, maugusto@nps.navy.mil.

part of the participants, and a clear understanding of the semantics of the messages exchanged. EDI implementations, despite their "standardized" nature vary dramatically from business to business [6].

2.1.3 Open Grid Services Architecture (OGSA) for Distributed Systems Integration

OGSA builds upon the concepts and technologies from the Grid and Web Services communities. It [1] defines standard mechanisms for creating, naming, and discovering transient Grid service instances; provides location transparency and multiple protocol bindings for service instances; and supports integration with underlying native platform facilities. It aligns the Grid technologies with the WS technologies, in particular the WSDL, to provide mechanisms required for creating and composing sophisticated distributed systems, including lifetime management, change management, and notification. OGSA has adopted Globus Toolkit as the underlying Grid technology solution.

Each of these above mentioned approaches have specific objectives and are, aimed typically at particular application domains. In the next section, two other approaches, Web Services and UniFrame that are generic in nature are discussed.

3 The UniFrame and Web Services Nexus

3.1 UniFrame Overview

The main focus of UniFrame is to provide a comprehensive framework for the software realization of distributed computing systems. It consists of (a) a meta-model for components and associated hierarchical set-up for indicating contracts and constraints of the component, (b) an automatic generation of glue and wrappers, based on designer's specifications to achieve interoperability, (c) a formal mechanism for precisely describing the meta-model, and (d) the formalization of the notion of the quality of service of each component and an ensemble of components.

3.2 Web Services (WS) Overview

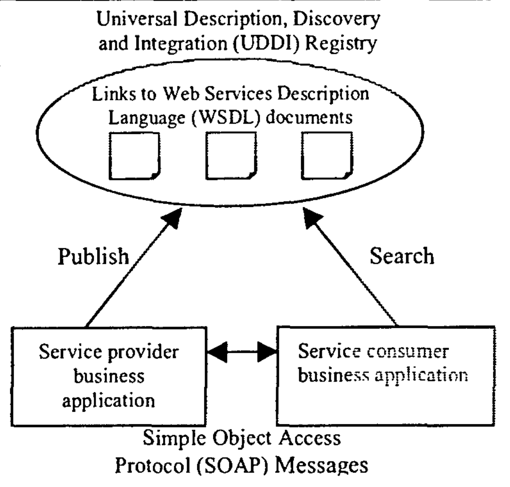
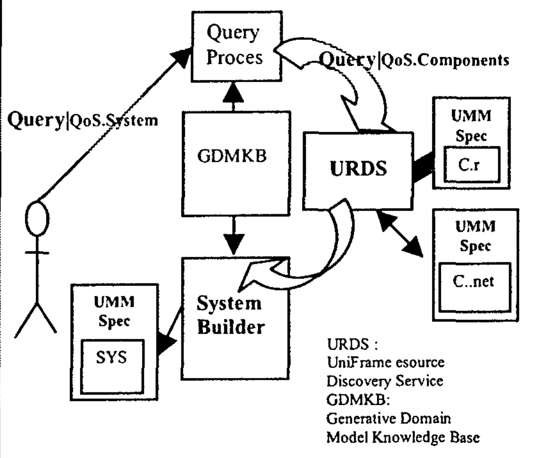
WS are based on existing protocols and technologies and provide a greater flexibility with respect to the interoperability, the reuse and the development of applications in a distributed environment. The underlying idea behind WS is to promote the "software as a service" paradigm. The use of open standards enables interoperability between components. These standards are based on XML, which enables WS to communicate with other applications in a programming language-, programming model-, and system software-neutral manner. XML forms the basis of the three standards: SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery and Integration) [5].

The next section indicates a comparison between the architectural aspects of the two frameworks, i.e., UniFrame and Web Services, and then the section 4 describes a model-based comparison.

3.3 Architectural Comparison

The following table shows the architectural comparison between the two paradigms:

	WEB SERVICES FRAMEWORK	UNIFRAME
OBJECTIVE	To provide a set of related standards which allow building of dynamic, loosely coupled systems composed of services, not bounded to any implementation and can be published, described, located and invoked over a network, more generally World Wide Web	To create a comprehensive framework that unifies the existing and emerging distributed component/service models under a common meta-model that enables the discovery, interoperability, and collaboration of components via generative software techniques [3,7]

GENERAL ARCHITECTURE	 Universal Description, Discovery and Integration (UDDI) Registry Links to Web Services Description Language (WSDL) documents Publish Search Service provider business application Service consumer business application Simple Object Access Protocol (SOAP) Messages	 Query Proces Query|QoS.Components Query|QoS.System GDMKB URDS UMM Spec C.r UMM Spec C..net UMM Spec SYS System Builder URDS : UniFrame esource Discovery Service GDMKB: Generative Domain Model Knowledge Base
BASIC TASKS/ PROCEDURES INVOLVED	▪ Service Development and Deployment (leveraging different platforms to one standard of web services using different Web Services development tools and software provided by vendors) ▪ Formal description of services (WSDL) ▪ Registration of services with UDDI (publish) ▪ Discovery of services (Find) ▪ Binding with the Service (Bind)	▪ Developing components using different current and future object models, such as, Java-RMI/CORBA/.Net/Web Services ▪ Informal and formal UMM (Unified Meta-Component Model) specifications of each component ▪ Querying the UniFrame for the system with desired Quality of Service parameters ▪ Creation of an integrated system made out of discovered components ▪ Incorporation of necessary glue and wrappers for QoS measurements and interoperation ▪ Checking to see if the test results of the integrated system satisfy criteria or not ▪ Refine Query or select alternate components to re-build and retest the integrated system
SERVICE/ COMPONENT DEVELOPMET DEPLOYMENT	▪ Development using frameworks that support them (e.g. .NET) or using different object models, which are then leveraged as services using the toolkits that support the technology ▪ Registering Services with the UDDI public/private registry	▪ Components are developed using different standard object models ▪ Deployment also under the same model with extra infrastructure provided by UniFrame to support seamless interoperation and system generation
DESCRIPTION OF SERVICES/ COMPONENTS	Web Service Description Language Document (WSDL file – XML)	UniFrame Meta-Component Model Description (UMM Specifications – informal text and XML)
DISCOVERY	Discovery through the UDDI Business or private registries (static registries)	Discovery through an search process involving active entities – headhunters and active registries [UniFrame Resource Discovery Service (URDS) Framework]
INTER-OPERABILITY OF SERVICES/COMP ONENTS	XML (standard for data exchange) and SOAP (Simple Object Access Protocol)	Automatic generation of glues and wrappers
SYSTEM INTEGRATION	▪ A hand-crafted approach wherein the responsibility of integration lies with the application developer by means of APIs of the WS ▪ Need to incorporate WS interfaces and	A comprehensive model-based approach forms the backbone of the system integration process right from the initial stages. The model follows an architecture-centric, domain-based and a technology-independent approach. The process

	integration capabilities within the existing "application integrating" tools and products	may be manual, completely automatic or a mix of both
RELIABILITY OF THE COMPOSED SYSTEM	Reliance on a third party (Web Service Auditors) which guarantees the reliability of a web service on basis of testing and certification during its creation as well as operational stage	Reliability based on test cases and formalism and a strong mathematical foundation of event traces and two level grammar
ADVANTAGES **ADVANTAGES (Contd..)**	▪ Builds upon open text-based standards (XML), thus aiding in interoperability ▪ Less additional cost involved in adoption, since employs existing infrastructure (Internet) and applications can be repackaged as Web Services	▪ No requirement of additional software tool to build components ▪ Automatic generation of glues and wrappers ▪ Quality of Service validation and assurance through event traces and formal domain knowledge; backed by a mathematical foundation ▪ Use of aspect-oriented programming to weave in the notion of QoS into the framework distinguishes UniFrame ▪ Active search process involving the notion of "headhunters"
LIMITATIONS	▪ Relatively new; standardization in progress, hence, Web Services created with current tools will not be compatible with the future technologies ▪ Use of text-based standards, XML, for communication may affect performance in some critical applications ▪ No standardized methods devised for assuring and validating Quality of Service; Use of third party "web service auditors"	▪ No standardization reached yet ▪ Experimentation and performance evaluation at a large scale and in a realistic domain not complete

Some of the important points tabulated above are described in detail in the next few sections.

3.3.1 Discovery Services

Web Services Discovery Process: The term discovery refers to the process of locating "Web Services" by means of registries. This process is carried out by businesses searching for services offering specific functionalities. WS Registries and Brokerages facilitate the discovery process and enable interactions between the service providers and requesters. The discovery process is classified into two categories [4]:

- Direct Discovery: This involves obtaining data from a registry, which is maintained by the service provider itself.
- Indirect Discovery: This involves obtaining data about a Web Service from a registry, which is maintained by a third party.

A service provider publishes the WSDL document containing the description of its Web Service, with the UDDI, which makes locations of such WSDL files available to a service requester. The Service Requester searches the UDDI based on certain criterion, such as functionality or a Quality of Service (QoS) attribute. Once it discovers a service, meeting its needs, it knows the method of accessing the Web Service by means of the WSDL file. It can now communicate with the Web Service directly via SOAP messages.

There are a few other discovery technologies, which support the discovery of Web Services apart from the UDDI specifications – ebXML and WS-Inspection for example. A Service developer/organization can combine these technologies with UDDI in order to take advantage of the features of both. For example, UDDI currently does not support a security model whereas ebXML does and so an organization can advertise its services through UDDI, on the other hand store its trading agreements and contracts through ebXML.

UniFrame Resource Discovery Service (URDS) Framework: The URDS architecture [8] provides a mechanism for an automated discovery and the selection of components meeting necessary QoS requirements. URDS is designed to act as a Discovery Service wherein new services are dynamically discovered while providing clients with a directory style access to services. The discovery process in URDS is "administratively scoped", i.e., it locates services within an administratively defined logical domain – in UniFrame a domain refers to industry specific markets such as Financial Services, Medical domain and Manufacturing Services, etc. The URDS infrastructure consists of two parts: (a) the Internet Component Broker (ICB) and (b) Headhunters.
The ICB, in addition to performing the functions of a conventional broker, also ensures the authentication of the principals of the system (Headhunters and Active Registries); cooperates with other ICB's deployed on the network to provide matchmaking between service producers and consumers; and acts as a mediator between two components adhering to different component models. A Headhunter is equivalent to a binder or trader in other models. However, unlike the trader, here the onus of registering components lies with the headhunter and not on the components themselves. Hence, the headhunter is capable of detecting the presence of service providers on the network, register the functionality of these service providers and return a list of service providers, which matches the requirements of the consumer requests forwarded by the Query Manager, to the ICB. The services are discovered by means of Active Registries (discussed later), with which the services are registered. The discovery process employed could vary from standard search techniques to broadcasts and multicasts to specific machines.

3.3.2 Service Descriptions

Web Service Description Language (WSDL) Document: It is an XML document for describing WS as a set of endpoints operating on messages containing either document-oriented (messaging) or RPC-payloads. Service interfaces are defined abstractly in terms of message structures and sequences of simple message exchanges and then bound to a concrete network protocol and data-encoding format to define an end-point. Related concrete end-points are bundled to define abstract end-points (services). The WSDL is extensible to allow description of end-points and the concrete representation of their messages for a variety of different message formats and network protocols [4].
UniFrame Meta-Component Model (UMM) Description: In UniFrame, components are autonomous entities. The UniFrame description of a component is more comprehensive and specified in a natural language-like manner. It indicates the functional (i.e., computational, cooperative and auxiliary aspects) and non-functional (i.e., QoS constraints) features of the component. These specifications are then refined into a formal specification based on the theory of Two-Level Grammar (TLG) and natural language specifications [9]. TLG specifications allow for a multi-level interface for the component. These levels are: Syntactic, Behavioral, Synchronization and QoS.

3.3.3 Registries/Repositories

Web Services Registries: The Web Services framework supports two kinds of repositories - UDDI and WS Brokerages.
UDDI: The UDDI standardization provides for "searchable Web Services Registries" which facilitate the storage, discovery and exchange of information about businesses and their Web Services. UDDI is implemented in two forms:
UDDI Business Registry: publicly accessible and maintained by Microsoft, IBM, Hewlett Packard and SAP.
UDDI Private Registry: accessible only to authorized users.
The various entities involved during the utilization of UBR (UDDI Business Registry) [4] are:

Operator Nodes: The organizations that host the implementation of the UDDI Business Registry are Microsoft, IBM, SAP, and Hewlett Packard. UBR operates on the principle of "register once and publish everywhere". This in turn implies a replication of the data within the operator nodes so that all instances of records are identical with each node. Operator nodes synchronize their information at least every 12 hours.

Custodian: The custodian for a company is the operator node with which it publishes its web services. A company can register and update its information only through its custodian. This prevents multiple versions of the data from entering in the four different operator nodes.

Registrar: These organizations do not host implementations of the UDDI but act as assistants for organizations in creating data (such as business and service descriptions) and publishing in the UBR.

Structure and Information Model of UDDI: XML forms the basis of the overall information structure of UDDI which can be broadly divided into following information levels:

White Pages: General information about the provider, such as its name, contact information and identifiers.
Yellow Pages: Categorization of the providers' information based on their services.

Green Pages: Technical information about the provider's services or products. Usually contains references to the WSDL documents of the services enabling the client to know as to how to interact with the Web Service.

UDDI supports certain APIs for the clients to use the registry. These include:
Publishing API – It supports the publish operation on the UDDI Registry. The access to this API is restricted to authorized users only. Operator nodes implement a form of Authentication protocol to allow legal organizations to access this API. By means of publishing API, an organization is able to execute commands to create and update information in its operator node.
Inquiry API: Supports the find operation in three different patters (browse, drill-down and invocation). This API is accessible to any individual on the UBR who wishes to locate a service or a kind of service.

WS Brokerages: The WS brokerages are web sites that house information about the available WS in the form of a list, along with their web addresses. These brokerages can also supply additional services, which can include advanced search capabilities based on category, organization name or schema type, service monitoring and service support, which can include services-related resources such as a tool that validates WSDL documents. Examples of some of the current Web Services Brokerages are: Allesta Web Service Agency, SalCentral Service, Xmethods and serviceFORGE.

UniFrame Registries: In the case of UniFrame, the entity that houses the information about components developed using a particular model is local to that component model. This entity is named "Active Registry", and is an enhanced version of the native registry of the corresponding object model. It has features such as *Activeness (*an ability to listen to multicast messages), *Introspection and a Capability to detect failures of the Headhunters.*

The conceptual difference that exists between registries of the two frameworks is in the way the registries participate in the discovery process of the components. In the case of the WS framework, the onus of locating components lies in the hands of the service requesters. While in UniFrame, the emphasis is on the automated discovery process provided by means of the URDS. Whether an organization needs to deploy one active registry per machine or one per many, is not decided and could vary depending on the size and necessity of the organization. While a service requester and publisher has to confirm to the underlying implementation of the UDDI registry as preferred by the company hosting it, the Active Registry is not as rigid and constraint since it builds upon the same native technology used for the development of components registered with it.

3.3.4 Quality of Service Assurances

Quality of Service Assurances in Web Services: Currently, service providers typically employ third parties to audit their web services during the creation stage as well as for reevaluation of the service on regular basis. An *auditor* achieves this in the form of testing and certification. Auditors may also be employed by the service requestors in order to gain a kind of guarantee about the level of service offered by the Web Service. The entire scenario employs "Service Level Agreements (SLA)" [4]. These are "legal contracts in which a service provider outlines the level of service it guarantees for a specific Web Service". When customers purchase the Web services subscription, they receive the services according to the quality-related contents specified by the SLAs. The service developer may maintain the SLAs. As the contents of the SLA are determined by the participating entities, there are no formal guidelines to specify the level of service a particular Web Service provides. The QoS requirements, which SLAs of WS's outline, include a*vailability, accessibility, integrity, performance, reliability, conformance to standards and security.*
Quality of Service framework of UniFrame: The approach followed by UniFrame can be stated as: building a precise model of the system's behavior (based on event traces) and then providing a programming formalism to describe the computations over these event traces. These are then applied in order to define different kinds of QoS metrics. UniFrame's iterative approach to system assembly from components meeting user's query specifications is based on constructive calculations of QoS metrics on representative set of test cases.
Quantifying the quality of service of the individual Commercial Off The Shelf (COTS) components, which compose to form an integrated system with a predictable quality, is one of the critical part of the UniFrame Approach. UniFrame provides a QoS Framework [5] for selecting, specifying and validating the QoS of components. The features of the UniFrame QoS framework are:

- An existence of a QoS catalog [10] containing detailed descriptions about QoS attributes, their classifications, their evaluation methodologies and the interrelationships with the other attributes.

- An integration of QoS at the individual component and distributed system levels.
- The validation and assurance of QoS, based on the concept of event grammars [11].
- An investigation of the effects of component composition on QoS; involving the estimation of the QoS of an ensemble of software components given the QoS of individual components.
- A QoS-centric iterative component-based software development process to ensure that the end product matches both the functional and QoS specifications.

UniFrame takes a domain-based approach in the classification and the discovery of components. Since every domain has its own constraints with respect to the QoS attributes, the QoS catalog aims to act as a checklist for any component developer/user interested in identifying and validating QoS attributes.

3.4 Model-based Comparison

- WS are all about XML and it being a text-based standard implies delays involved in parsing it, which may prove vital in performance-critical applications. XML uses two sets of redundant tags to mark up every piece of information it represents. The tags are usually written to be humanly readable, which makes the actual tags a lot longer than they need to be. Also, one character in a Unicode document can be up to four bytes. Four bytes in some other proprietary binary format used by technologies such as DCOM or RMI can hold a lot more information than just one character. The ability to serialize the data over a connection, parse it quickly and efficiently is what plays a vital role in applications interacting over the network [12]. UniFrame, on the other hand, leverages the components in a way so that they are a part of an application while remaining within their own object-model. This allows for more efficient ways of electronic communication.
- HTTP is the preeminent protocol to transfer WS content and is allowed a free access through firewalls. HTTP, although used almost everywhere because of its reliability and ubiquity, is also not the most efficient transport protocol [12]. HTTP relies on a constant connection between the client and server when a request is made. This constant connection causes an overhead in cases when the data that needs to be transferred is quite small. However, in the WS's universe, many transactions are essentially asynchronous. This in turn implies that the response of a web service request is not guaranteed. HTTP was not meant to deal with this kind of asynchronicity. It also relies on only one side initiating communication and the other side only responding to the request. This approach inhibits true peer-to-peer exchanges through Web services. A newer version of HTTP aims to fasten communications by making use of compression, but some of the previous issues still need to be pondered upon. Other protocols such as SMTP, over which Web Services can be implemented, still do not provide a major breakthrough in this respect. As UniFrame does not attach itself to a specific protocol, it avoids some of the drawbacks related to the usage of HTTP.
- The only guarantee that a service requester has about a Web Service is through its SLA. No other explicit mechanisms are mandatory in the WS world. Thus, the user of WS may or may not have a mechanism to validate the QoS claims made by the creator of WS. Hence, a requester can terminate its contract if the WS's fail to deliver what it claimed in its SLA. In contrast, UniFrame makes the notion of quality explicit during the creation of components. It also provides the user means (by the use of event grammars, glues and wrappers) to validate the QoS of any component made available by a supplier.
- In the world of B2B, Web Services prove to be a major benefit since they provide the needed flexibility and ability to operate across the Internet on completely disparate systems owned by completely independent entities. However, in EAI solutions, the major drivers are not only interoperability but also speed and efficiency, and with those requirements, Web services don't really seem to meet the need. Organizations globally are becoming aware of the importance and need of integration across disparate platforms. An organization with numerous applications needs EAI solution and corporations that are extending their processes with partners need B2B. The future holds potential for a solution set that provides the functionality for both the requirements frameworks. The UniFrame with its unbiased approach is an attempt in this direction.
- Although UDDI registries, both public and private, offer a great deal of advantage in terms of an application integration of the participating companies, they have their own set of limitations too. Firstly, because UDDI is fairly new, it has not reached standardization in a complete way, which holds true for UniFrame as well. Secondly, the UDDI Business Registry poses the question of data reliability. UniFrame does not involve the notion of publicly accessible registries. The Active Registries only allow authorized entities to publish components and interacts with the headhunter, thereby reducing the threats of data compromise. The discovery mechanism of the UMM Framework involves the headhunter storing the data about the components after it retrieves it from the Active Registries. The duration of the time interval after which this process repeats itself can be controlled so as to guarantee the freshness of the data within the meta-repository of the headhunters.

UDDI registries, although describe web services, do not evaluate them [4]. It does not house the Quality-of-Service information about a web service and requires an extensive search on the service-consumers part to do so. UniFrame on he other hand, provides an extensive Quality-of-Service framework to do so.

3.5 Integrating Web Services into UniFrame

As outlined above, the WS and UniFrame differ in their approaches and associated implementation techniques. However, they can complement each other to provide solutions for future distributed systems. UniFrame uses the Generative Domain Model [13] to describe the properties of domain-specific components and to elicit rules for assembling heterogeneous components. One possible approach to integrate WS in UniFrame could be to use WS as a mechanism to wrap heterogeneous components. Due to the open nature of WS, such an approach will ease the task of assembling heterogeneous components adhering to existing and new object models. Furthermore, since WS are weak in representing the business semantics of application domains, this will also lead to the enrichment of WS technology in terms of semantic representation by following a model driven approach for specific domain-specific component models. UniFrame can then automatically generate WSDL from the models with the help of generators.

4 Conclusion

Developing component-based software solutions for distributed systems is an inherently complex task. Any approach to tame these complexities must account for disparities that exist due to the existence of different object models. Web Services and UniFrame are two approaches that propose effective solutions for future component-based distributed systems. In this paper, an analysis of these two approaches has been presented. Although these two approaches differ from each other, they can also complement each other and provide a comprehensive solution for the creation of distributed systems. The proposed approach to integrate Web Services into UniFrame needs further investigation and is being currently explored.

5 References

[1] Foster, I., Kesselman, C., Nick, J., Tuecke, S., The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[2] World-Wide Web Consortium (W3C), "Web Services Activity", 2002, http://www.w3.org/2002/ws.

[3] Raje R., Bryant B., Auguston M., Olson A., Burt C., 2001, "A Unified Approach for Integration of Distributed Heterogeneous Software Components," Proceedings of the 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration, pp. 109-119.

[4] Dietel, H., Dietel, P., DuWaldt, B., Trees, L., Web Services – A Technical Introduction, 2003, Prentice Hall, Upper Saddle River, New Jersey 07458

[5] Dhingra, V., "Business-to-Business Ecommerce," http://projects.bus.lsu.edu/independent_study/vdhing1/b2b.

[6] A Darwin Partners and ZapThink Insight, "Using Web Services for Integration", http://www.xml.org/xml/wsi.pdf

[7] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., Burt, C. C., 2002, A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components, *Concurrency and Computation: Practice and Experience*, vol. 14, pp. 1009-1034.

[8] Siram, N. N., Raje, R. R., Olson, A. M., Bryant, B. R., Burt, C. C., and Auguston, M., An Architecture for the UniFrame Resource Discovery Service, Proceedings of the 3rd Int. Workshop Software Engineering and Middleware, Springer-Verlag Lecture Notes in Computer Science, Vol. 2596, 2002.

[9] Bryant, B. R. and Lee, B.-S., "Two-Level Grammar as an Object-Oriented Requirements Specification Language," Proceedings of the 35th Hawaii International Conference on System Sciences, 2002, http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf.

[10] Brahnmath, G. J., Raje, R. R., Olson, A. M., Auguston, M., Bryant, B. R., Burt, C. C., A Quality of Service Catalog for Software Components, Proceedings of the 2002 Southeastern Software Engineering Conference, pp. 513-520.

[11] Auguston, M., Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars, Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering, 2000, pp. 159-166.

[12] Hudson, M. J., The Web Services Placebo, http://www.intelligententerprise.com/020917/515e_business1_1.shtml

[13] Czarnecki, K., Eisenecker, U.W., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

A Component Assembly Approach Based On Aspect -Oriented Generative Domain Modeling

Fei Cao, Barrett R. Bryant, Carol C. Burt [1]

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL, USA

Rajeev R. Raje, Andrew M. Olson [2]

Department of Computer and Information Science
Indiana University Purdue University at Indianapolis
Indianapolis, IN, USA

Mikhail Auguston [3]

Computer Science Department
Naval Postgraduate School
Monterey, CA, USA

Abstract

We present an approach towards automatic component assembly based on aspect-oriented generative domain modeling. It involves the lifecycle covering the component specification generation, and subsequent assembly of implementation components to produce the final software system. Aspect-oriented techniques are applied to capture the crosscutting concerns that emerge during the assembly process. Subsequently, those concerns are woven to generate glue/wrapper code for assembling heterogeneous components to construct a single integrated system.

Key words: Component Assembly, Generative Programming, Generative Domain Model, Component Specification, Aspect Orientation, UniFrame, Two-Level Grammar.

[1] Email: {caof, bryant, cburt}@cis.uab.edu
[2] Email: {rraje, aolson}@cs.iupui.edu
[3] Email: auguston@cs.nps.navy.mil

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

As software component development technology becomes more mature, the notion of developing software systems by assembling Commercial-Off-The-Shelf (COTS) components (implemented in models such as COM[4], DCOM[5], EJB[6], CCM[7]) becomes not only theoretically rational, but also practically sound. Component-Based Software Composition offers a development paradigm with reduced time-to-market and cost while achieving enhanced productivity, quality and maintainability [3].

But component assembly remains mainly either a handcrafting effort or proprietary approach [28]. It is becoming an even harder problem when components are delivered in binary form which may need binary code adaptation [17], or when the underlying implementation language, deployment environment are heterogeneous. For the latter case, what's commonly seen is a middleware approach such as CORBA, that allows the components to work cooperatively across language and platform boundaries. However, this approach may also add extra complexity that makes the construction of a distributed system more difficult [9].

UniFrame[8] is a framework for seamless interoperation of heterogeneous distributed components. It aims to automate the process of integrating heterogeneous components to create distributed systems that conform to quality requirements. By automatic generation of glue/wrapper code based on the developer's functional and non-functional specification ([2], [26]), the system generated will be tailored to specific requirements as opposed to being a monolithic end product, and reliability is also enhanced. In this paper, we present an approach to support the automatic component assembly in UniFrame by applying aspect-oriented generative domain modeling. In Section 2, we introduce the background information of UniFrame. Section 3 and 4 present our approach of using aspect-oriented generative domain modeling for component assembly. Section 5 presents some discussion, followed by the description of related work together with the conclusion in Section 6.

2 Background

2.1 Generative Programming

As is introduced in [10], *Generative Programming (GP) is a software engineering paradigm based on modeling software families such that, given a particu-*

[4] Component Object Model, http://www.microsoft.com/com
[5] Distributed Component Object Model, http://www.microsoft.com/com/tech/dcom.asp
[6] Enterprise Java Beans, http://java.sun.com/products/ejb
[7] CORBA® (Common Object Request Broker Architecture, http://www.omg.org/corba) Component Model, http://www.omg.org/cgi-bin/doc?orbos/99-07-01
[8] Unified Framework for Seamless Integration of Heterogeneous Distributed Software Components - http://www.cs.iupui.edu/uniFrame

lar requirement specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge. The requirement specification is sometimes referred to as *ordering* of products; the terminology used to specify family members is referred to as the *problem space*; the implementation components with their possible configurations form the *solution space.* The problem space and solution space, together with the associated configuration knowledge, constitute the *Generative Domain Model* (GDM) [10]. The distinct property of GP is that it is not only about a development *for reuse* in terms of building a GDM for software system families, but also about a development *with reuse* in terms of using GDM to generate concrete systems; it focuses on generation of system families rather than a one-of-a-kind system.

2.2 UniFrame

With advances in network technology, software systems are shifting from a closed, centralized architecture to being open and distributed; from being homogeneous in implementation to adopting heterogeneous components for constructing the whole system. To harness the omnipresent components in a distributed system while having to address the inherent complexity of such a paradigm, the functional and non-functional properties of components must be formally captured, and there needs a means to assure the specified QoS (Quality of Service)[9] for the system assembled from components. UniFrame is a framework to address those concerns [26]. It uses a Unified Meta-component Model (UMM) [25] to encode the meta-information of a component such as functional properties, implementation technologies, and cooperative attributes.

In UniFrame, a GDM is also used to capture the domain knowledge and to elicit assembly rules. But the use of a GDM doesn't include the implementation components: this part is assumed to be offered in a distributed system environment by different vendors observing the stipulated specifications in the problem space of the GDM; those implementation components are exposed by vendors and are subject to location by a distributed resource discovery service [27]. In addition, the GDM in UniFrame is used to capture the assembly rules for the discovered components.

Figure 1 illustrates the big picture of UniFrame. The annotated number represents the processing order. Starting from domain experts, a GDM will be created (1.1) and will be used together with some domain standards (1.2) as guidelines (2.1, 2.2) for component developers to implement components in solution space. Those implementation components, after being quantified with some QoS parameters (3), will be exposed to a distributed resource dis-

[9] In this paper, "non-functional aspect", "non-functional-property" and "Quality of Service (QoS)" may be used interchangeably.

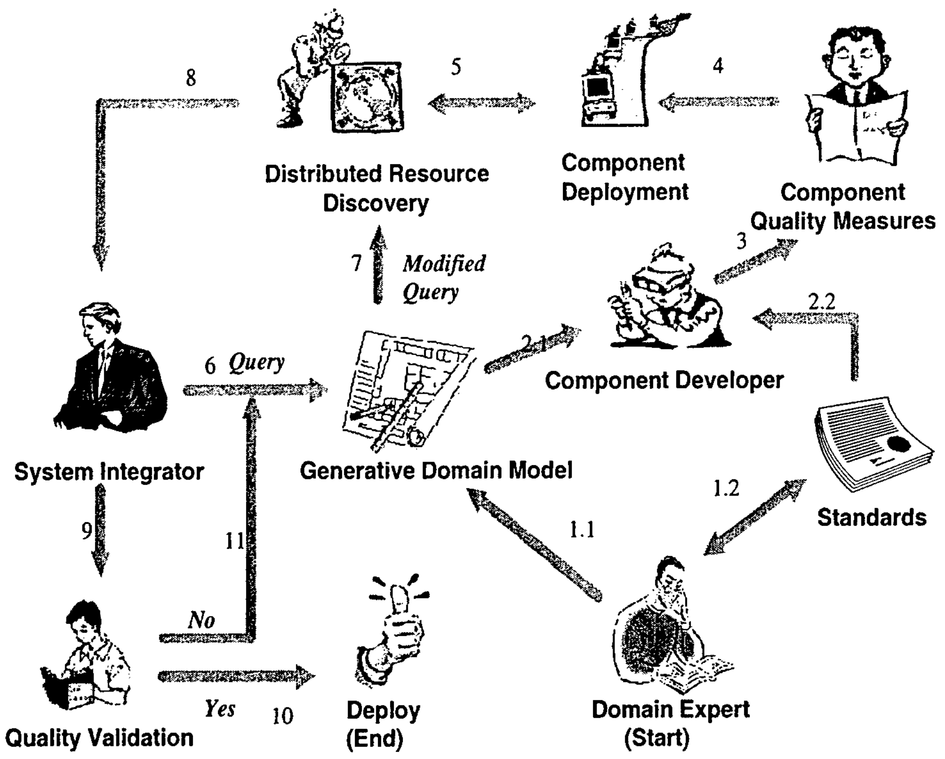


Fig. 1. the Process of UniFrame.

covery service (5). Thereafter, a system integrator will query into the problem space of the GDM for available/deployed component information (6), and then command the resource discovery service (7) to fetch the required components (5,8) for assembly. The component assembly is subject to validation (9) based on specified QoS requirements. If it is not validated (11), then the integrator has to initiate the query and integration process iteratively. As it can be seen from above, the GDM stands as a crucial part of UniFrame, and how GDM is represented so as to facilitate the component assembly is of vital importance. We call the means to represent GDM *generative domain modeling*, which is further detailed in the next section.

3 Overview of the Approach

3.1 Specification of Components in the Solution Space of the UniFrame GDM

Components in UniFrame are specified using the formalism of Two-Level Grammar (TLG) [4]. The specification in TLG provides flexibility in translating TLG specifications to other representations, such as other formal specification languages like the Vienna Development Method [21], or application code [6]. TLG contains two context-free grammars, one describing type domains and the other describing rules and operations on those domains. Note it is not required to have both levels. Below is a template TLG specification.

```
class Identifier-1
   Identifier-1, Identifier-m1 ::  DataType1; DataType2;...;
      DataType-n1.
   Function-signature-1,...Function-signature-m2 :
      function-call-1,function-call-2,..., function-call-n2.
end class Identifier-1.
```

The line containing "::" denotes the first-level type domain definition, for which the right hand side of "::" provides the type (which is called a *meta-type*) while the left hand side provides the variable name. Note the right hand side may specify multiple types at the same time, which are delimited by ";". The left hand side may also have multiple variables separated by ",", which are of the same meta-type as defined on the right hand side. Also note the meta-type may form a hierarchy (*meta-type hierarchy*). For example, *BankOperation* may be the meta-type of the *Withdraw* operation, while Service may be the meta-type of the *BankOperation.* Consequently, *Service* is also regarded as the meta-type of *Withdraw.*

The line containing ":" denotes the definition of the second-level rule/operation (also called *hyper-rule*) over the first-level type domains. ';' can be used in the right hand side of ":" to delimit multiple rules which share the same function signature on the left hand side. Note both first-level and second-level may contain multiple (including zero) sentences as opposed to just one sentence of each in the above description.

3.2 Separation of Concerns in Generative Domain Modeling

Consider the following two component specifications in the GDM problem space (note this simple example serves for the motivation purpose only–full definition of a component description language is provided in Section 4.2).

```
Component BankServer
   provides AccountManagement:
      applies AccessControl
end Component

Component BankClient
   requires AccountManagement:
      uses RMIServer applying QoSMonitor
end Component
```

In the *BankServer* specification, the provided service *AccountManagement* uses *AccessControl.* But as business rules are subject to change, the BankServer may lift the AccessControl or enforce other type of controls, either

Functional	Business rule enforcement
	Specific technology instrumentation
	Pre/post condition
	...
Non-Functional	Profiling
	QoS Validation
	QoS Instrumentation
	...

Table 1
Assembly Related Aspects

of which will reduce the reusability of the original BankServer implementation component. In the *BankClient* specification, the "RMIServer" and "QoSMonitor" that are required for a server-side AccountManagement service represent the glue/wrapping logic needed to integrate the client and server components. This tangles the BankClient component and also reduces its reusability as glue/wrapping requirements change.

Aspect-Oriented Programming (AOP) [18] provides a means to capture crosscutting aspects in a modular way with new language constructs, and also provides a *join point model* to "hook" the aspects with the base program. This is the basis of augmenting the component specification approach with aspect orientation in order to separate those crosscutting assembly-related aspects of components. Those aspects do not need to be implemented by vendors. The separation will refine the granularity of GDM, and contribute to the *maximal combination, minimal redundancy, and maximum reuse*, which are the desired properties of implementation components [10] in the solution space of GDM. Consequently, the component assembly process evolves into an aspect weaving process. Table 1 provides the tentative catalog of assembly related concerns.

Figure 2 illustrates the aforementioned idea. The arrow ending with a diamond figure represents the *include* relationship as in the standard UML[10] notation. Separation of concerns [23] is introduced into the domain analysis phase, the output of which is the GDM. The GDM includes the concerns identified at the domain analysis phase (which are also called *early aspects*[11]), and those aspects are collectively stored into a repository called the *aspect library*. This aspect library corresponds to the configuration knowledge as indicated in Section 2.1. The GDM also includes Component Description Language (CDL, the actual definition to be provided in Section 4.2) in its problem space part; the CDL is also used as a guideline for implementation

[10] Unified Modeling Language, http://www.omg.org/uml
[11] http://early-aspects.net/

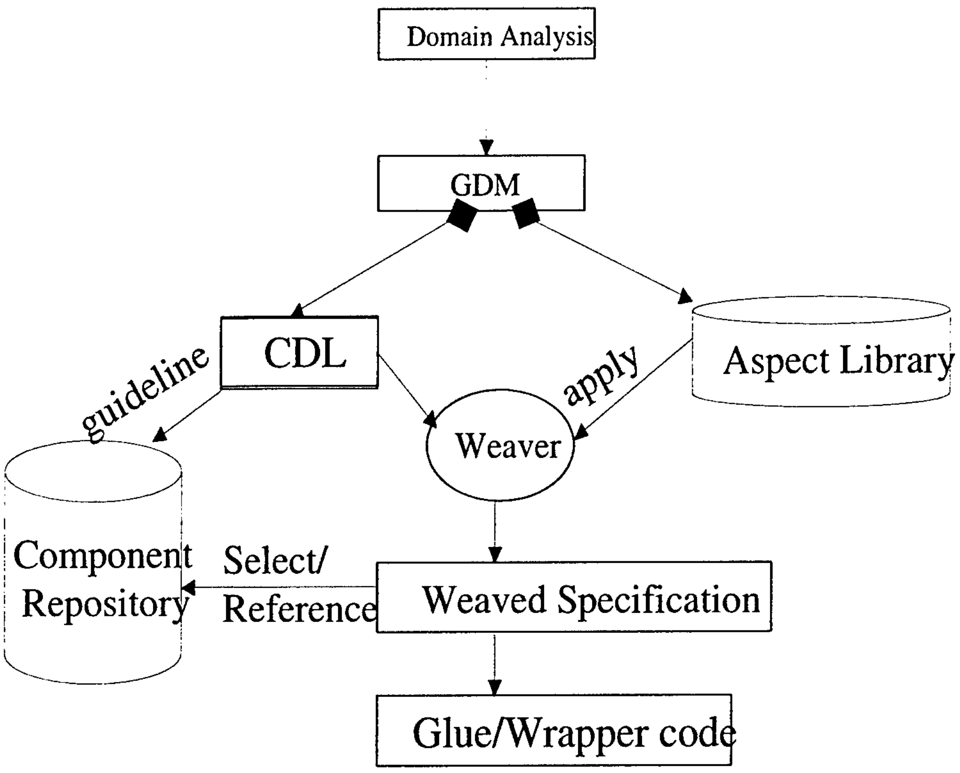


Fig. 2. Aspect-Oriented Generative Domain Modeling

of components by different vendors. Upon an *ordering* request over the GDM problem space, the CDL in the problem space will be weaved with involved assembly aspects into the specifications for glue/wrapper code generation, which by referencing the implementation components, will be used to generate final glue/wrapper code to connect the components.

4 Multi-Stage Component Assembly

Before we detail the component assembly process in Section 4.3, we provide the related specification definitions in Section 4.1 and 4.2.

4.1 Definition and Use of Aspect

In such AOP languages as AspectJ [19], aspects are defined in a way that is closely bound to the base program (the *join point* is specified syntactically based on the base program). In contrast, in Figure 2, aspects are separately stored as a library. Thus, a *join point model* is required to hook the aspects to the targeted program so as to apply the related *advice* provided in the aspect.

Aspect Description Language (ADL)is defined as follows:

```
aspect <aspectname>
   advises:  <Meta-type>.
```

```
    [before:  <advice>.]
    [after:  <advice>.]
  end aspect <aspectname>
```

The name enclosed in "<>" represents a grammar variable, which will be exemplified in Section 4.3. The "[]" is used to delimit a part that is optional. Those notations apply to the following Aspect Usage Language (AUL) and CDL as well. The <Meta-type>, which is defined as in Section 3.1, is used to specify the types of domain services that this aspect can be applied. The *advice* following the directive *before/after* provides the pre/post actions to be performed or pre/post conditions to be enforced before/after the domain services, which can be used for temporal dependency specification and tracing/QoS code instrumentation. For example, in [30], before/after advice is used to specify rules for model checking. Consequently, the aspect library represents a collection of assembly rules.

AUL is defined as follows:

```
  apply <aspectname> on <type> [when <relational-expression>]
```

<aspectname> corresponds to an assembly-related aspect, which already provides a means to specify assembly rules as described in the preceding paragraph. The <type> has to be consistent with the applicable <metatype> in the ADL of <aspectname>. By *consistent* we mean the <metatype> as in the ADL of <aspectname> should reside at the root position of some meta-type hierarchy (see Section 3.1 for definition), where <type> is part of the hierarchy. The *when* directive in AUL further specifies the scenarios using relational expressions, under which this aspect can be applied; in addition to the base-program oriented weaving such as in AspectJ [18], the advice quantification [12] here is also user-case oriented. It's quite straightforward that AUL can be used in *product ordering* specification as indicated in Section 2.1. Note the definitions of ADL and AUL are inspired by [11], where non-functional aspects are separated from components themselves to increase the component (and non-functional aspect) reuse, and the non-functional aspects are handled with similar language constructs as ADL and AUL described here.

4.2 Component Description Language (CDL)

CDL is used in the problem space of GDM to specify the components, their required and/or provided services in a way to achieve *maximal combination, minimal redundancy, and maximum reuse* (as mentioned in Section 3.2) as the result of aspect-oriented generative domain modeling. CDL is defined in TLG as described in Section 3.1.

```
component <componentname>
   <DomainVariable1>,..<DomainVariable-m> ::
      <DomainType-1>; <DomainType-2>;; <DomainType-n>.
   [requires <Domain-Specific-Service>:
      function-call-11, function-call-12,, function-call-1n.]
   [provides <Domain-Specific-Service>:
      function-call-21, function-call- 22,, function-call-n.]
end component  <componentname>
```

The first level of CDL provides the type-hierarchy of domain variables. The *requires/provides* specification constitutes the second level. For the *requires* specification, the right-hand side details the requirements; for the *provides* specification, the right-hand specification further specifies the semantics of the provided services.

4.3 Aspectual Component as a Paradigm of Component Assembly

The Aspect Library as shown in Figure 2 captures the *general* business and technology requirements in terms of assembly-related concerns, and a single AUL expression addresses a single concern. In contrast, a component captures groups of behaviors and component assembly captures groups of concerns with mixed scenarios. *Aspectual Component* is used here to address the group of concerns occurring in the component assembly scenario.

The concept of aspectual component [12] is firstly proposed in [22], for which aspects are decoupled from the base program by being defined as a generic aspectual component, which is instantiated later over a concrete data-model using a *connector* construct. Examples of aspectual components and connector specifications will be provided in the following section. The concept of aspectual component fosters the integration between AOSD (Aspect-Oriented Software Development) and Component-Based Software Development (CBSD) ([8], [29]). The aspectual component model will also be used here for component assembly. However, the original *aspectual component* is in Java, while here it is a language-independent specification in TLG. The connector specification classifies server components' related services into a category based on meta-type. The *connector specification* also includes related operations associated with the meta-type. The meta-type can be regarded as one kind of *join point* in AOP, while the related operations in the connector specification provides *advice.* The meta-type in an aspectual component is the basis upon which client and server component get hooked up; the join point model to be used is again type-based as in Section 4.1.

We integrate the ideas into an process diagram in Section 4.3.1, which is reified by an example in Section 4.3.2.

[12] Note in our own context, the definition of aspectual component is subject to adjustment over its original definition in [22].

4.3.1 The Overall Picture

Figure 3 provides the multi-stage component assembly process. Stage 1 is mainly about the introduction of the GDM (from domain analysis), which includes CDL in problem space and Aspect Library as configuration knowledge. Stage 2 involves the weaving of the aspect specification into the component specification for each component involved in the assembly process. Stage 3 illustrates the process of the component assembly specification generation based on the aspectual component model. This stage involves a *connector repository*, where the connector specifications will be registered, and the aspectual component will initiate a *query* into the connector repository to find the matching connector specification based on meta-type consistency, and to apply the associated advice thereafter. The connector specification is translated from the CDLs of the server component (service provider) and the aspectual component specification is translated from the client component (service consumer). After the full assembly specification is generated, by referencing the component repository (which stores the set of component UMM specifications retrieved by the discovery service in UniFrame), glue/wrapper code will be generated in the final step.

4.3.2 An Example

To help clarify the aforementioned process, a simple example is provided below, demonstrating how the *aspectual component* approach can be adapted to the component assembly process. Assume that the component A is a banking domain client component written in Java RMI requesting some banking service from a server. Below is the partial specification of A's CDL:

```
A.0 Component A
A.1 BankOperation::  Service.
A.2 Bank::BusinessDomain.
A.3 Platform::TechDomain.
A.4 requires BankOperation:  Platform= ''RMI''.
A.5 end Component A.
```

Below is an ADL for a QoS measurement aspect stored in the Aspect Library and AUL to use that aspect.

```
aspect QoSMeter
   advises: BankOperation.
   before: EventTrace.setBeginTime().
   after: EventTrace.setEndTime().
end aspect QoSMeter

apply QoSMeter on A.BankOperation.
```

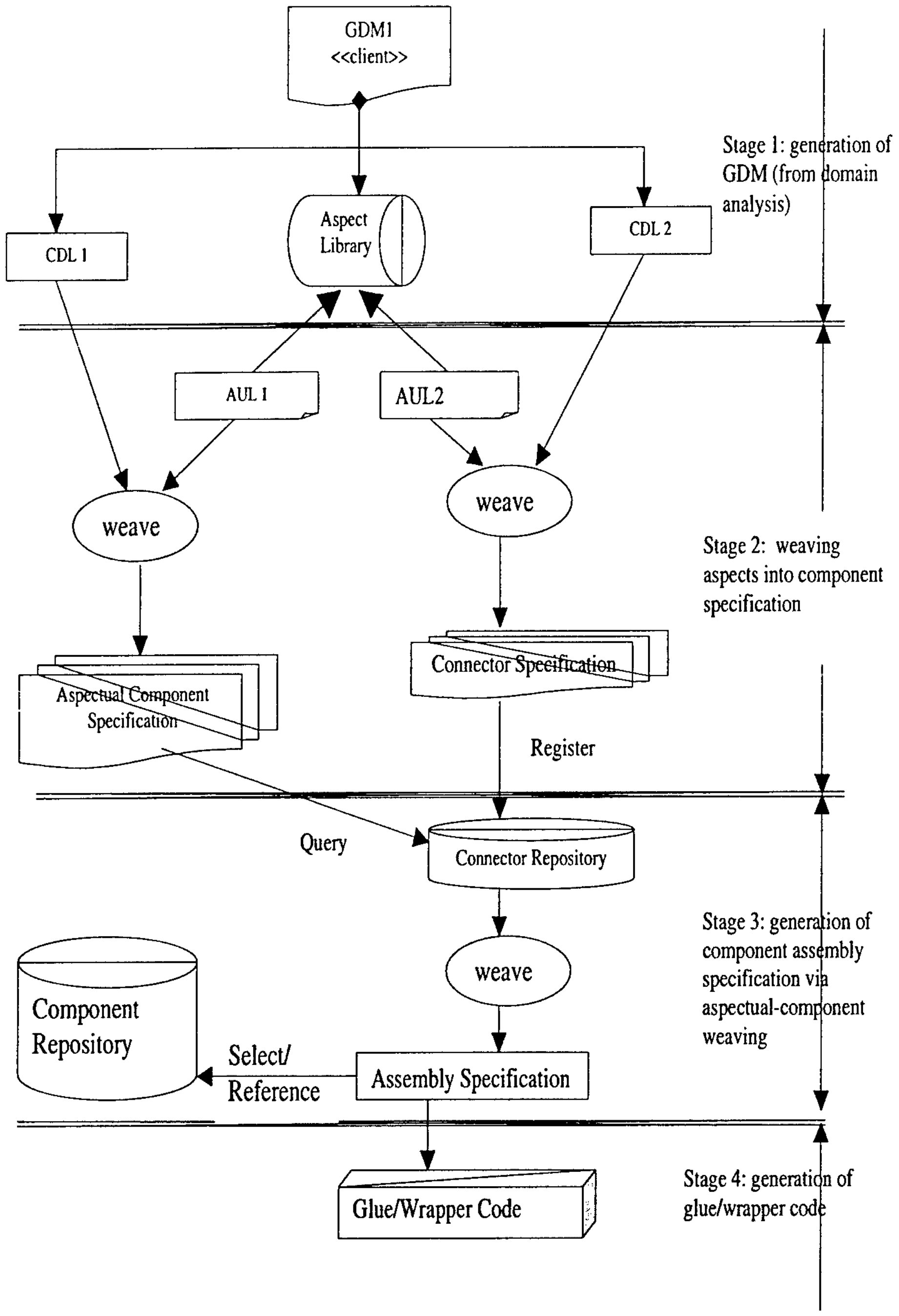

GDM1
<<client>>
Aspect Library
CDL 1
CDL 2
Stage 1: generation of GDM (from domain analysis)
AUL 1
AUL2
weave
weave
Stage 2: weaving aspects into component specification
Aspectual Component Specification
Connector Specification
Register
Query
Connector Repository
Stage 3: generation of component assembly specification via aspectual-component weaving
weave
Component Repository
Select/ Reference
Assembly Specification
Glue/Wrapper Code
Stage 4: generation of glue/wrapper code

Fig. 3. Multi-Stage Gluing/Wrapping

The above specification of component A weaved with QoSMeter aspects will be translated into the following aspectual component specification:

```
B.0 AspectualCom A
B.1 Bankoperation::  Service.
B.2 Bank::BusinessDomain.
B.3 expect Bankoperation.
```

```
B.4 expect wrap Argument.  //usage interface
B.5 replace Bankoperation://modification interface
B.6       EventTrace.setBeginTime(),
B.7       expected().wrap(<<Platform=''RMI''>>),
         //each <<...>> corresponds to each
         // expression in right hand side of '':''  of A4
B.8       EventTrace.setEndTime().
B.9 end AspectualCom A
```

B.6 and B.8 are weaved from the QoSMeter aspect representing client-side concerns. Note those lines prefixed by *expect* denote operation signatures that are expected to be supplied with *advice* (which actually corresponds to server-side services requested), and the expect-directive corresponds to the *join points* in AOP. Expected operations are either used (usage interface) or modified (modification interface, preceded with *replace*) in the aspectual component definition. This process is similar to that described in [22].

Assume the component B is a banking domain server component implemented in CORBA providing some banking services.

```
C.0 Component B.
C.1 Withdraw, Deposit::  Port;Bankoperation.
C.2 Bank::Domain.
C.3 Platform::TechDomain .
C.4 provides Bankoperation:  Platform= ''CORBA''.
C.5 end Component B.
```

Note in line C.1, the two types denoted in the right hand side of "::" means both withdraw and deposit are not only *Port(s)* (which means they are banking services offered to external components), but also *Bankoperation(s)*.

Below is an ADL for an Access Control aspect [5] from the Aspect Library.

```
aspect AccessControl
   advises: Service.
   before: Log.Check().
end aspect AccessControl
```

This aspect can be applied to any *Service* (meta-type, thus applicable to *Withdraw*). Consequently, before each call to Service, Log.Check() will be called to verify the credentials.

The following specification will be translated from the component B specification with the AUL of the preceding aspect AccessControl.

```
D.0 connector A-B
D.1 {B.Withdraw, B.Deposit} is BankOperation. //join points
```

```
D.2 wrap(Argument):
D.3         apply AccessControl on B.WithDraw, B.Deposit,
D.4             apply RMIAspect on BankOperation when
D.5                 Argument.getname (''Platform'')==''RMI''
D.6 end connector A-B
```

Note that lines D.2-D.5 further implement the *advice* part for the join points (here, *Withdraw* and *Deposit* operations). The body of *wrap* is to wrap the BankOperation with RMI specific code. This is similar to [24], in which CORBA related operations are modularized as aspects and then woven into application code to derive a CORBA implementation. The difference here is that, those RMI or CORBA related aspects are pre-built and retrieved from the aspect library, and they are represented with high-level specifications (in ADL) rather than at the application code level. Upon weaving in Stage 4, the *wrap* routine in the connector specification will be weaved into the aspectual component specification.

The example illustrated in this section shows that assembly-related concerns (functional and non-functional) of two components can be handled in separate modules (here in the aspectual component definition and connector specification) from the component specification itself. ADL and AUL provide leverage for the assembly process itself to be easily specified and managed. Consequently the assembly can be implemented by using a weaver to weave assembly-specific advice together with component specifications.

5 Discussion

UniFrame, the motivating project of the component assembly approach presented here, aims at automating the process of integrating heterogeneous components to create distributed systems that conform to quality requirements. Generative Programming (GP) is the underpinning solution to fulfill this vision. In order to realize the vision of GP for the highest level of automation, during the domain engineering phase, the creation of the domain model may be applied using Model Integrating Computing (MIC) [20], which is a technology for using domain-specific modeling and a model based generator to compose systems of various forms. MIC has been applied to create a Generic Feature Modeling Environment (GFME) [7] to model system families and generate reusable assets automatically. Based on the component assembly approach presented in this paper, Table 2 describes generative programming in UniFrame.

Also note the assembly paradigm described in Section 4.3 follows a client/server architecture, whereby the client component (service consumer) specification is translated into the aspectual component specification. In the event the components to be assembled are not following that kind of architecture, the *ordering* specification itself may be translated into an aspectual component

Generative Programming	*UniFrame*
Feature modeling	GFME
Components are generated in domain implementation phase	Components are implemented by vendors. Generation only occurs at system level
Configuration Knowledge	Aspect Library
Mapping of problem space to solution space	Resource Discovery Service to search components based on component specification
Domain Specific Language (DSL)	CDL, AUL, ADL
Generator	Aspect Weaver

Table 2
Generative Programming in UniFrame

specification, and then the assembly process as shown in Figure 3 can be applied.

6 Related Work and Conclusion

Recently, there has been work on the application of AOSD to CBSD. One notable work is the aspectual component [22] as described in Section 4.3, which provides a language approach to the effort of reusing aspects. The aspectual component model is adjusted and used here for component assembly. Grundy further introduces the notion of *Aspect-Oriented Component Engineering* (AOCE) ([13], [14], [15], [16]). The aspects in AOCE have a broad definition, which include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, data management, component inter-relationships, and configuration characteristics. AOCE, as an engineering approach, covers the lifecycle of component engineering, from component requirements and specification, to implementation, deployment, and testing. In contrast to AOP, which highly relies on *code weaving*, AOCE aims to use aspect-codified capacities to support component provisions and requiring of aspect-related services in a general way. In this sense, AOCE can be applied for building the aspect library. None of the related work ever considers applying AOSD to assist the component assembly, however.

This paper presents an approach to apply aspect orientation in the generative domain modeling phase and then leverage the aspect weaver to help component assembly, in particular, for assembling components of client/server architecture. Two repositories (aspect library, connector repository) are used, which aligns with the distributed component assembly style. A type-based join point model is used which can efficiently decouple the aspect definition and aspect usage to promote the reuse of aspects. Compared with the invasive composition approach as described in [1], we weave the assembly-related concerns toward ultimately generating stub/skeleton code for gluing/wrapping components, while the original components (which represent the business logic core), together with their references to stub/skeleton code, will not be affected. This is necessary for black-box components which do not allow invasive methods.

Future work includes the evolution of the aspect library, the application of MIC to domain engineering to automatically generate CDL, and the development of the weaver to weave CDL and ADL. The implementation of glue/wrapper code generation based on the generated assembly specification using the UMM specifications of discovered components must also be integrated into this process.

7 Acknowledgements

We'd like to acknowledge the anonymous reviewers for their helpful suggestions. This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

References

[1] Aßmann, U., "Invasive Software Composition," Springer-Verlag, 2003.

[2] Brahnmath, G. J., Raje, R. R., Olson, A. M., Auguston, M., Bryant, B. R., Burt, C. C., *A Quality of Service Catalog for Software Components*, Proceedings of the Southeastern Software Engineering Conference ((SE)^2 2002), pp. 513-520, April, 2002.

[3] Brown, A. W., "Large-Scale Component-Based Development," Prentice Hall, 2000.

[4] Bryant, B. R., Lee, B.-S., *Two-Level Grammar as an Object-Oriented Requirements Specification Language,* Proceedings of 35th Hawaii International Conference on System Sciences, 2002, http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf.

[5] Burt, C. C., Bryant, B. R., Raje, R. R., Olson, A. M., Auguston, M., *Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control,* Proceedings of 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003), pp. 159-171, September, 2003.

[6] Cao, F., Bryant, B. R., Burt, C. C., Raje, R. R., Auguston, M, Olson, A. M., *A Translation Approach to Component Specification,* OOPSLA '02 Companion, pp. 54-55, November, 2002.

[7] Cao, F., Bryant, B. R., Burt, C. C., Huang, Z., Raje, R. R., Olson, A. M., Auguston, M., *Automating Feature-Oriented Domain Analysis,* Proceedings of 2003 International Conference of Software Engineering Research and Practice (SERP 2003), pp. 944-949, June, 2003.

[8] Choi, J. P., *Aspect-Oriented Programming with Enterprise JavaBeans,* Proceedings of 4th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2000), pp. 252-261, September, 2000.

[9] Colyer, A., Blair, G., Rashid, A., *Managing Complexity in Middleware,* Proceedings of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), March, 2003.

[10] Czarnecki, K., Eisenecker, U. W., "Generative Programming: Methods, Tools, and Applications," Addison Wesley, 2000.

[11] Duclos, F., Estublier, J., Morat, P, *Describing and Using Non Functional Aspects in Component Based Applications,* Proceedings of 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), pp. 65-75, 2002.

[12] Filman, G., Friedman, D., *Aspect-Oriented Programming is Quantification and Obliviousness,* Proceedings of OOPSLA Workshop on Advanced Separation of Concerns, pp. 168-177, October, 2000.

[13] Grundy, J. C., *Multi-perspective Specification, Design and Implementation of Components using Aspects,* International Journal of Software Engineering and Knowledge Engineering, 10(6):713-734, December 2000.

[14] Grundy, J. C., *An Implementation Architecture for Aspect-oriented Component Engineering,* Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 249-256, June, 2000.

[15] Grundy, J., Patel, R., *Developing Software Components with the UML, Enterprise Java Beans and Aspects,* Proceedings of the 2001 Australian Software Engineering Conference, pp. 127-136, August 2001.

[16] Grundy, J. C., Ding, G., *Automatic Validation of Deployed J2EE Components Using Aspects,* Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), pp. 47-58, September 2002.

[17] Keller, R., Hölzle, U., *Binary Component Adaptation,* Proceedings of European Conference on Object-Oriented Programming (ECOOP'98), Springer-Verlag, LNCS 1445, pp. 307-329, 1998.

[18] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J., *Aspect-Oriented Programming,* Proceedings of European Conference on Object-Oriented Programming (ECOOP'97), Springer-Verlag, LNCS 1241, pp. 220-242, 1997.

[19] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W., *An Overview of AspectJ,* Proceedings of European Conference on Object-Oriented Programming (ECOOP'01), Springer-Verlag, LNCS 2072, pp.327-353, 2001.

[20] Ládeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J. and. Karsai, G., *Composing Domain-Specific Design Environments,* IEEE Computer, 34(11):44-51, 2001.

[21] Lee, B.-S., Bryant. B. R., *Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language,* Proceedings of ACM Symposium on Applied Computing (SAC 2002), pp. 932-936, 2002.

[22] Lieberherr, K., Lorenz, D., Mezini, M., *Programming with Aspectual Components,* Technical Report, NU-CCS-99-01, 1999, http://www.ccs.neu.edu/research/demeter/papers/aspectual-comps/aspectual.ps.

[23] Parnas, D., *On the Criteria To Be Used in Decomposing Systems into Modules,* Communications of the ACM, 15(12): 1053-1058, December 1972.

[24] Pulvermuller, E., Klaeren, H., Speck, A., *Aspects in Distributed Environments,* Proceedings of Generative Component-based Software Engineering (GCSE 99), Spinger-Verlag, LNCS 1799, pp. 37-48, September 1999.

[25] Raje, R., *UMM: Unified Meta-object Model for Open Distributed Systems,* Proceedings of 4th IEEE International Conference of Algorithms and Architecture for Parallel Processing (ICA3PP 2000), pp. 454-465, 2000.

[26] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., Burt, C. C., *A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components,* Concurrency and Computation: Practice and Experience, 14(12):1009-1034, 2002.

[27] Siram, N. N., Raje, R. R., Auguston, M., Bryant, B. R., Olson, Burt, C. C., A. M., *An Architecture for the UniFrame Resource Discovery Service,* Proceedings of 3rd International Workshop on Software Engineering and Middleware (SEM 2002), Springer-Verlag, LNCS 2596, pp. 22-38, 2002.

[28] Sutherland, J., Heuvel, W.-J. v. d., *Enterprise Application Integration and Complex Adaptive Systems,* Communications of the ACM, 45(10):59-64, October, 2002.

[29] Suvée, D., Vanderperren, W., and Jonckers, V., *JAsCo: an Aspect-Oriented approach tailored for Component-based Software Development,* Proceedings. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp. 21-29, March, 2003.

[30] Ubayashi, N., Tamai, T., *Aspect-Oriented Programming with Model Checking,* Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), pp. 148-154, April, 2002.

QoS-UniFrame: A Petri Net-based Modeling Approach to Assure QoS Requirements of Distributed Real-time and Embedded Systems[1]

Shih-Hsi Liu
Barrett R. Bryant
Jeffrey G. Gray
Univ. of Alabama at Birmingham
Birmingham, AL 35294, USA
{liush,bryant,gray}@cis.uab.edu

Rajeev R. Raje
Andrew M. Olson
Indiana Univ. Purdue
Univ. Indianapolis
Indianapolis, IN 46202, USA
{rraje,aolson}@cs.iupui.edu

Mikhail Auguston
Naval Postgraduate School
Monterey, CA 93934, USA
maugusto@nps.navy.mil

Abstract

Assuring quality of service (QoS) requirements is critical when assembling a distributed real-time and embedded (DRE) system from a repository of existing components. This paper presents a two-level approach for assuring satisfaction of QoS requirements in the context of a reduced design space for DRE systems. A dynamic and parallel approach is introduced to prune off the infeasible design spaces at the first level. Evolutionary algorithms cooperating with a domain-specific scripting language then discard less probable design spaces using statistics. These techniques fulfill the collective objectives of pruning and assuring the design space at system assembly time.

1. Introduction

Distributed real-time and embedded (DRE) systems are widely used in military, manufacturing, and control systems [17]. Many of these systems consist of legacy components. From the perspective of software engineering, there is an urgent demand to fulfill the need of the development, evolution and integration of DRE systems from existing components. This is in the vision of the UniFrame project [16]. During the synthesis of a DRE system, various appropriate components can be selected from a repository. However, numerous design and deployment decisions for the selected components usually generate a tremendous number of possible alternatives for constructing a DRE system. The design information (i.e., specific design and deployment decisions and information of involved components) required for synthesizing a DRE system is called a *design space* [13]. Among the huge number of possible design spaces, many of them, in fact, do not satisfy the requirements of the DRE system (i.e., *constraint satisfaction*). In addition, constructing a DRE system (e.g., an avionics system) is naturally expensive and less modifiable. In order to decrease the possibility of errors occuring after construction of a DRE system, validating a DRE system in advance is also necessary to conserve the future potential costs. Therefore, it is necessary to have a formal, manageable, scalable and automatic design space exploration approach to prune unsatisfactory design spaces (i.e., unsatisfactory assembled cases), and to validate the rest of the assembled cases of a DRE system from its requirements at system assembly time.

In addition to functional requirements, quality of service (QoS) that pertains to the usage of resources is an important requirement of DRE systems. *QoS parameters* are used to evaluate the degree of performance of QoS using utility functions, which is the mathematical formulas that show the utility of QoS. For example, timeliness is a quantifiable QoS parameter that estimates whether the deadline is met by the addition of the execution time of involved components. Security, however, is a non-quantifiable QoS parameter that evaluates the level of security of a DRE system being achieved with a user-defined function. This paper presents a two-level assurance technique, called "QoS-UniFrame," for QoS of DRE systems assembled from components. This technique, based on artificial intelligence and statistics, reduces the design space and validates QoS requirements at system assembly time. Consequently, we believe that discarding infeasible and less probable cases at system assembly time will require less runtime validation. In addition to assurance and validation, QoS-UniFrame concentrates on observing and adapting non-orthogonal QoS parameters (e.g., CPU usage and throughput) seldom addressed by researchers. QoS-UniFrame also exploits AspectJ [8] to promote reusability and modularity by separating the source code to analyze constraints from that to construct design spaces. The modification of the constraint analysis code is convenient and isolated from the rest of the source code.

[1]This research was supported in part by U. S. Office of Naval Research award N00014-01-1-0746.

This paper is organized as follows: in the next section, background and related work are addressed; section 3 introduces the framework and techniques of QoS-UniFrame; section 4 provides a case study; finally, we conclude and point out the future work of the paper in section 5.

2. Background and Related Work

2.1 Background of QoS-UniFrame

The implementation of QoS-UniFrame is based on two techniques described in the following subsections.

2.1.1 Petri Nets

System engineers need to make various decisions while constructing a DRE system. Different decisions may require cooperation with different components. For one decision, there may be diverse execution orders, execution time, and events to trigger execution among the chosen components. Therefore, there are a huge number of possible assembled cases generated based on different decisions and components with the consideration of various orders, time, and events. QoS-UniFrame reduces the complexity of exploring all possible assembled cases for building a DRE system by evaluating their QoS requirements. The evaluation of QoS of a specific assembled case depends on *when, what*, and *how* the components request QoS requirements. *When* expresses the specific time or before/after a specific event a component has effect on a QoS parameter; *what* specifies which QoS parameter is inspected; *how* represents the relationship of data access among the components.

In most QoS research (e.g., [13]), dataflow analysis is applied to explore possible solutions for assurance of QoS requirements. *A segment of a dataflow* is a directed arrow between two (sets of) components generated by a single decision. The directed arrow means that two (sets of) components have requests to access a QoS parameter from one to another, or have effect on a QoS parameter by cooperation between each other. For multiple decisions after a specific segment of a dataflow, multiple segments will be generated and flow to corresponding (sets of) components. Finally, various *dataflows* (also called *QoS systemic paths*), and the sequences of the segments of dataflows, will be generated as a tree structure by different decisions. Namely, the leaves of the tree are all possible assembled cases created based on different decisions. However, the dataflow analysis is not sufficient for analyzing DRE systems, because some QoS analyses require additional information. For example, in some DRE systems, the performance of the systems relies on the levels of QoS to be achieved. Different levels of QoS will trigger corresponding events, and vice versa. Furthermore, time and priority constraints also influence QoS. All of these characteristics show the difficulty for dataflow analysis to assure QoS requirements of DRE systems.

A *Petri Net* is a formalism similar to dataflow analysis, but has additional abstractions beneficial in modeling concurrent and asynchronous systems [14]. It is expressed by a *Petri Net graph*, which is a visual representation that can model a DRE system. A Petri Net graph consists of abstractions adequate to analyze QoS requirements of possible assembled cases of a DRE system. *Tokens* represent QoS parameters with the identifiers, and the types and ranges of the parameters. *Places*, (sets of) components in a DRE system, are the same as the starting and end points of a segment of a dataflow in the dataflow analysis. *Flows*, same as dataflows, control the flowing direction of the QoS parameters. *Transitions* embody associated predicates and functions for time, priorities and event triggers to determine *what, when* and *how* QoS parameters are to be processed [14]: only when specific conditions are satisfied can the QoS parameter be processed by descendent components.

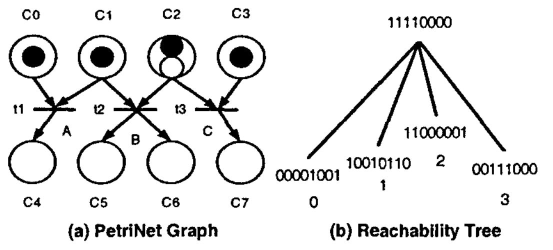


Figure 1. The Petri Net graph and its reachability tree example.

To explore various possible assembled cases, the *reachability tree* is exploited to diagnose a Petri Net graph. Figure 1 (a) is a simple Petri Net that shows the formalism to model a DRE system with various design decisions and time and event concerns described below. Assume that eight components (C0 to C7) constitute a simple DRE system. Both C1 and C2 have two decisions such that C1 can either work with C0 or C2, and C2 can cooperate with C1 or C3. A QoS parameter (black token) that processes C1 and C2 will be accessed by both C5 and C6. C4, C5 and C6, and C7 can deal with the QoS parameter at time *t1, t2* and *t3*, respectively. C1 and C2 with two flows means the token will stream to one of two transitions without preference (i.e., alternative decisions). Finally, transition B and C verify if C2 has an event (gray token) execution that triggers C5 and C6 to access the QoS parameter. For B, three conditions cause the black token stream to C5 and C6: the black tokens in C1 and C2 are both flowing in; the gray token in C2 is flowing in, and is verified by B; and timer is at time *t2*. Consequently, one assembled case is made, and branch 1 of Figure 1 (b) is constructed correspondingly. Figure 1 (b) is the reachability tree of Figure 1 (a) generated by the construction principles stated above. The purpose of a Petri Net is to explore and generate possible assembled cases by its reachability tree based on the design decisions, selected components considering priorities, events, and time.

There are several advantages to modeling DRE systems using Petri Nets. First, as stated before, Petri Nets' abstractions and characteristics are appropriate to simulate DRE systems, either for functional or nonfunctional requirements. They overcome the insufficiency of the dataflow analysis. In addition, the transitions regarding priority, time, and events infer the concept of dynamic decision making such that only when a specific transition is persuaded can an assembled case by the decision be generated.

2.1.2 AspectJ

AspectJ [8] is an aspect-oriented programming (AOP) language [9] for Java. It provides a modular mechanism to avoid the error-prone, fragile and tedious modification work for constraint analysis. An *aspect* recognizes the points of the method crosscutting Java's classes using *pointcuts*, and then defines how the modification should be made using *advice*. The aspect code is *weaved* into the Java base code with good modularity such that any change of the modification is isolated in the aspect. Hence, AspectJ promotes a better means to modularize and reuse the source code. QoS-UniFrame exploits AspectJ to recognize the methods of the reachability tree construction, and insert the constraint analysis method code.

2.2 Related Work

An *Ordered Binary Decision Diagram (OBDD)* [2] applies symbolic representations (i.e., binary encodings) to prune off the unsatisfactory design spaces [13]. It encodes mode space (i.e., functional behaviors that QoS-UniFrame does not cover), configuration space (i.e., dataflow), and constraints into binary representations. Binary operations are used to compute the fulfillment of constraints. However, the OBDD method suffers from the following disadvantages. First, binary operations for addition and multiplication are rigid and not user-friendly. It is not easy for system analysts to adjust the evaluation of pruning design spaces adaptively. In addition, this binary method requires sufficient temporary variables for computation. Second, many of the QoS parameters are non-orthogonal such that adjustment of one QoS parameter may substantially affect other QoS parameters. It is hard to specify a composite non-orthogonal constraint by means of conjunction and disjunction. A quantitative expression (e.g., a linear or nonlinear function) would be a better alternative. Third, the OBDD representation is not mature enough to solve system-level constraint problems and "the scalability of the method becomes susceptible and results in an exponential blow-up in OBDD representation" [13]. Most importantly, OBDD is a static design space pruning approach such that the computation can be processed when a dataflow with corresponding constraints is entirely constructed. All of these disadvantages motivate the development of QoS-UniFrame.

There has been considerable research to validate scheduling requirements of DRE systems. In [3], the timing constraint is validated by a symbolic model checking approach. Symbolic model checking is an extension of model checking such that analysis is based on symbolic transition representation and propositional logic with the extension of time operators. In [4] and [6], specialized Petri Nets were applied to verify time behaviors of DRE systems. All assurance by either model checking or Petri Nets has an inherent problem that validation does not always guarantee that the actual synthesized DRE systems are perfectly satisfactory: unpredictable behaviors that sometimes occur in DRE systems degrade the confidence of validation. Therefore, supportive statistical references utilized by QoS-UniFrame will be valuable as unpredictable behaviors occur.

3 QoS-UniFrame

Before the details of QoS-UniFrame are addressed, a brief example is given to illustrate *why* and *how* QoS-UniFrame solves the design space exploration problem with the constraint satisfaction:

A water treatment plant requires deploying new treatment units (TUs) to two new water treatment pools. Under the limit of the budget, the system and deployment engineers would like to ascertain the best performance of collective TUs from the blueprint. During the system design stage, different design and deployment decisions are made such as the order and the priority of the TUs, and the locations of the specialized TUs. In addition, the deployment of the TUs has various restrictions such as the bandwidth and the signal strength of the wireless network, the life of a battery in each TU, and the processing speed of the CPU in each TU.

Numerous decisions and constraints require concentrations in this project, and many of them have mutual effects. Hence, a manual procedure to construct and manage this project is error-prone and tedious. QoS-UniFrame answers these requests to ease the workload of the design decisions with constraints of the project. Starting from functional and nonfunctional requirements, a use case scenario is analyzed to determine the static and dynamic QoS requirements. System engineers construct a visual Petri Net model according to their design and deployment decisions. The system engineers depict the mutual behaviors of each component based on their QoS parameters in the Petri Net model. System analysts write the AspectJ codes with respect to the evaluation of strict or orthogonal static constraints (defined later), such as the total capacity of the batteries of TUs. These aspects are weaved into a dynamic and parallel approach to generate a tree abstraction including all feasible cases. Backtracking and branch-and-bound algorithms are employed to prune off infeasible assembled cases based on strict or orthogonal static QoS requirements at the first level. System analysts then write a domain-specific scripting code of evolutionary algorithms. The source code takes non-orthogonal or non-strict static, and dynamic QoS (defined later) into account with specific mathematical functions. The evolutionary algorithms will generate statistical results automatically. The less probable cases will be eliminated according to the discarding policies written in the domain-specific scripting code. The survival cases will be stored back to the knowledge base with their statistical information. Figure 2 shows the framework of QoS-UniFrame.

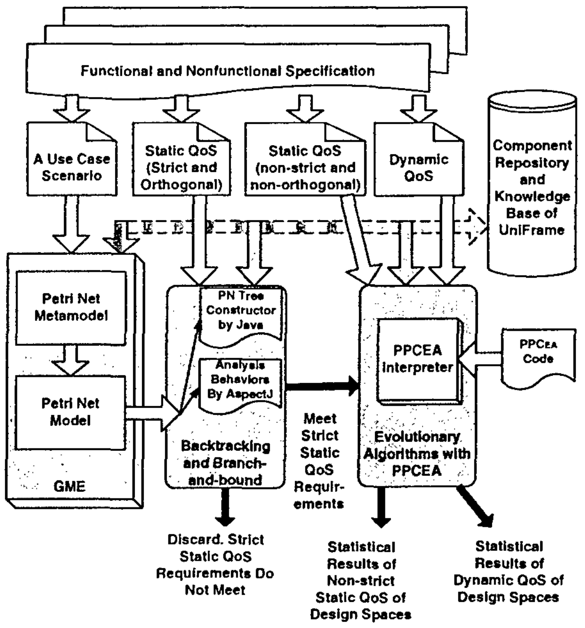

Functional and Nonfunctional Specification
A Use Case Scenario
Static QoS (Strict and Orthogonal)
Static QoS (non-strict and non-orthogonal)
Dynamic QoS
Component Repository and Knowledge Base of UniFrame
Petri Net Metamodel
Petri Net Model
GME
PN Tree Constructor by Java
Analysis Behaviors By AspectJ
Backtracking and Branch-and-bound
Meet Strict Static QoS Requirements
PPCEA Interpreter
PPCEA Code
Evolutionary Algorithms with PPCEA
Discard. Strict Static QoS Requirements Do Not Meet
Statistical Results of Non-strict Static QoS of Design Spaces
Statistical Results of Dynamic QoS of Design Spaces

Figure 2. The framework of QoS-UniFrame.

3.1 Classification of QoS Parameters

QoS-UniFrame currently concentrates on those QoS requirements that can be quantified. Namely, non-quantifiable QoS requirements (e.g., security and reliability) are out of our scope. QoS-UniFrame further classifies quantifiable QoS requirements into static and dynamic. *Static QoS* is design-related, and *dynamic QoS* is substantially influenced by the deployment environment. Many of the static QoS requirements can be evaluated at component assembly time, yet dynamic QoS requirements need either simulators or virtual machines to monitor, predict, and adapt the QoS concerns. However, several dynamic QoS requirements can be assessed by referring to a component's previous state and observations, as stored in a knowledge base at assembly time. Static and dynamic QoS parameters may be further subclassified into strict and non-strict, and orthogonal and non-orthogonal QoS. *Strict QoS* requirements (e.g., hard deadlines) force DRE systems to meet the requirements. Otherwise, the system will be incorrect because it cannot meet its QoS. *Non-strict* QoS requirements (e.g., soft deadlines) allow margins of error when meeting QoS requirements. The performance of the system will be degraded according to the magnitude that non-strict QoS requirements are not assured. *Orthogonal QoS* implies that its adaptation will not influence other QoS, yet *non-orthogonal QoS* substantially affects other QoS directly or indirectly. According to the hierarchy of classification, QoS-UniFrame separates static and dynamic QoS into a two-level assurance process.

3.2 Petri Net-based QoS Modeling

In order to explore design spaces efficiently and assure QoS requirements manageably, a formal approach to model and analyze the components of a DRE system with respect to its QoS is necessary: a Petri Net-based QoS modeling language has been created in the Generic Modeling Environment (GME) [10].

```
public aspect Analysis {
      pointcut Monitor(QosPar par) :
          call(public void *.createNode(..)) && args(par);
      after(QosPar par1) : Monitor(par1)
      {   double temp=0;
          if (par1.getName().equals("MPC"))     {
             //MPC stands for "Maximum Flow Processing Capacity"
             temp=par1.getValue();
            //evaluate MPC's QoS requirement  }
      }
      after(QosPar par2) : Monitor(par2)
      {   double temp=0;
          if (par2.getName().equals("BL"))      {
            //BL stands for "Battery Life"
            temp=par1.getValue();
            //evaluate BL's QoS requirement    }
      }
}
```

Figure 3. Constraint analysis method code for QoS parameters written in AspectJ.

As stated before, a Petri Net can explore and produce design spaces using the reachability tree. QoS-UniFrame evaluates strict or orthogonal static QoS requirements as a child node of a reachability tree is generated, and remove infeasible child nodes. Thus, strict or orthogonal static constraint analysis methods crosscut the source code of the child node construction of the reachability tree. The source code that analyzes constraints is written in AspectJ [8] as shown in Figure 3, and is weaved into the source code of the child node construction. In Figure 3, *pointcut* "Monitor" recognizes the method that generates a child node of the reachability tree. The first *after* advice statement evaluates the maximum flow processing capacity (MPC). It shows that after the "createNode" method is called, the QoS parameter is accessed, and then is evaluated by bounding and criterion functions (defined later). The second *after* advice statement evaluates the battery life (BL) using different bounding and criterion functions after the "createNode" method is called.

Implementing Petri Nets with GME and AspectJ contributes several merits. Because GME is a metaconfigurable modeling tool that permits customization [10], Petri Net models (i.e., simulation of DRE systems) can extend new features easily. Clear and appropriate syntactical and semantic design constraints supported in GME moderate the possibility of the errors occuring at the design phase. The visual modeling environment of GME also provides a user friendly and easily manageable environment for system engineers. In addition, separation of concerns of construction of QoS systemic paths and constraint analysis methods promotes reusability and modularity of source code. Various orthogonal QoS parameters can be evaluated concurrently by writing different advice in the analysis aspect (Figure 3). In this context, concurrency means that all of the constraint analysis codes are embedded in a child node construction method; namely, all advice crosscuts the same pointcut. Thus, it is necessary to define the advice precedence (i.e., weaving order of the advice) to avoid conflicts.

3.3 Backtracking and Branch-and-bound

In order to decrease the design spaces dynamically, the reachability tree construction code and its analysis aspect (Figure 3) are embedded into backtracking or branch-and-bound (B/B) algorithms [7]. The B/B algorithm that QoS-UniFrame exploits is the first level assurance to evaluate static QoS parameters that are strict and orthogonal, as in [13]. The *backtracking algorithm* employs a depth-first search on the reachability tree structure with bounding and criterion functions. Bounding functions are the constraints of strict and orthogonal static QoS requirements, and criterion functions (i.e., QoS utility functions) are used to determine the optimal solutions of a QoS systemic path, either maximal or minimal. The backtracking algorithm constructs the reachability tree from the root by depth-first search. It evaluates the bounding and criterion functions at every intermediate node. If the criterion applied to certain nodes does not meet the bounding function, the backtracking algorithm will stop generating all descendant nodes. Alternatively, the *branch-and-bound algorithm* operates with the reachability tree using various search algorithms. *LC-search* [7] is an improved search algorithm with a ranking function QoS-UniFrame chooses to implement. Similarly, the branch-and-bound algorithm traces from the root of a reachability tree. The ranking function determines the next node (i.e., live node) to be evaluated. LC-search intelligently ranks the live nodes to avoid the fixed order searches. Bounding and criterion functions in the backtracking algorithm play the same roles to stop constructing unsatisfactory child nodes. Therefore, the B/B algorithm *dynamically* eliminates the unsatisfactory design spaces based on strict and orthogonal static QoS requirements. Unlike most pruning design space approaches, such as [13], that evaluate one design space at a time, the B/B algorithm introduces a "parallel pruning concept" that cuts infeasible descendant leaves concurrently; namely, all the child nodes of an unsatisfactory intermediate node are discarded at the same time, which means infeasible design spaces are eliminated simultaneously.

3.4 Evolutionary Algorithms

In the DRE domain, it is tedious and time-consuming to validate one QoS requirement at a time. The B/B algorithm processes various strict and orthogonal static QoS parameters simultaneously writing different advice in an aspect. For non-strict or non-orthogonal static QoS requirements, and dynamic QoS requirements, QoS-UniFrame utilizes evolutionary algorithms (EAs) [12] as the second level assurance. An *EA* is a search and optimization technique based on the principles of natural selection and survival of the fittest [12]. The decision of the fittest (i.e., maximum, minimum or average) comes from the results of linear or nonlinear fitness functions in EAs. The fitness functions solve the tedious and time-consuming problem of non-strict static QoS, and the side effect problem of non-orthogonal (static and dynamic) QoS by combining all of the associated QoS requirements into a mathematical formula. Because dynamic QoS requirements need to comply with the deployment environment, QoS-UniFrame processes static and dynamic QoS requirements in separate steps. QoS-UniFrame has developed a domain-specific scripting language, called PPCEA [11], to make EAs expeditious and adaptable. PPCEA and AspectJ express the assurance of QoS requirements by means of linear or nonlinear functions. These representations make the assurance process easier to scale than the OBDD approach at system assembly time.

3.4.1 Static QoS Requirements

The B/B algorithm is, in fact, able to evaluate non-strict/non-orthogonal static QoS requirements by AspectJ. However, the unique purpose of the B/B algorithm is to remove infeasible design spaces with the dynamic and parallel concept. Hence, we postpone computing non-strict/non-orthogonal static QoS until the second level assurance. An EA evaluates the best results of non-strict/non-orthogonal static QoS parameters by a user-defined fitness function. For example, a DRE system constructed by a set of PDAs that meets battery maximum capacity may estimate the optimal solution of the lifetime, the disposal fee, and the purchase cost of the batteries by a fitness function. Therefore, a user-defined fitness function can satisfy this demand.

3.4.2 Dynamic QoS Requirements

Evaluating dynamic QoS requires the cooperation of the deployment environment. However, the statistical results of dynamic QoS by EAs at component assembly time may serve as excellent estimates and as substitutions as unpredictable behaviors occur later at runtime. EA solves the best, worst, and average fitness values and their standard deviations of a user-defined fitness function. Dynamic QoS requirement validation, such as deadlines for real-time systems, uses the previous state information of a component in the knowledge base to obtain the statistical results. Some assembled cases of these statistical results can be the references of runtime validation evaluation, and others may be eliminated by discarding policies invented based on PPCEA. User-defined discarding policies determine how and which assembled cases are rejected. More details will be explained in the next subsection.

3.4.3 PPCEA

To obtain the statistical outputs from EAs efficiently and to discard less probable assembled cases flexibly, a domain-specific scripting language, Programmable Parameter Control for Evolutionary Algorithms (PPCEA) [11], has been developed. PPCEA keeps the evolution process simple and raises the control parameter settings up to a high abstraction level in a programming fashion. In PPCEA, a configuration mechanism is provided to embed the parameters of EAs (e.g., crossover, mutation and discard rate, and population size) and its fitness function into the computation of EAs. The modification of these parameters is by a programming fashion, i.e., assignment statement. This mechanism provides the flexibility for users to find the optimal solution by different kinds of parameter settings [11].

```
genetic
   Discard := 1.1;  //discard rate by parameter tuning
   while (t <= 10) do
     init;          //initialize population
     call_EA;//evaluate fitness value for a population
     Temp := Temp + Worst;//Temp is temporary variable
     t := t + 1
   end;
   Temp := Temp / t;
   if (Temp > QoS*Discard)
     //Avg of Worst value far from requirement
     delete_gene //delete test cases not satisfied
   fi;
end genetic
```

Figure 4. Parameter tuning discarding policy written in PPCEA.

After defining the fitness function and parameters, PPCEA decides which genotypes (i.e., assembled cases) should be deleted from the population by the discarding policies with their discard rates. Users can apply parameter tuning, deterministic, or adaptive [5] discarding policies to the discard rate. Parameter tuning determines the value of the discard rate by assigning a constant value before each EA run. The deterministic method assigns the discard rate before the evaluation by a deterministic rule based on linear algebra [11]. Finally, the adaptive method adjusts the discard rate during the run of evaluation [11]. Figure 4 shows the example of parameter tuning discarding policy that operates with the discard rate. "t" is the counter for the while loop; "Discard" is the discard rate for discarding policy; "QoS" is a dynamic QoS requirement; "Worst" is the worst fitness value; "Temp" is the temporary variable for the convenience of computation; "call_EA" evaluates the values of fitness function of each genotype; and "delete_gene" discards those genotypes that do not meet the requirements. In Figure 4, if the average of ten worst cases is greater than 1.1 (i.e., user-defined discard rate) times the strict dynamic QoS requirement, the test case can be rejected.

4 A DRE System Case Study

This section presents a Petri Net-based QoS model of an example DRE system representing the water treatment plant described in section 3. The system engineers would like to examine the best performance of the water treatment ability under certain constraints:

(a) Due to the budget constraint, only three and two treatment units can be chosen for pools one and two for the water treatment process, respectively.

(b) the total maximum flow processing capacity is at least 50 million gallons per day.

(c) the battery life of each TU has at least 15 hours left.

(d) total CPU usage is at most 70 percent.

(e) total water treatment volume of selected TUs is at least 35 million gallons per day.

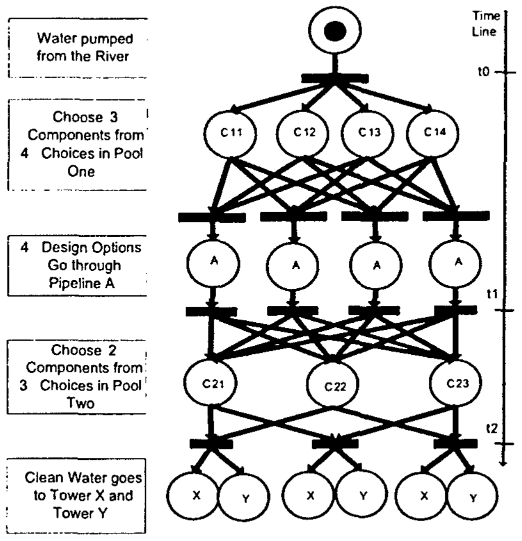
(f) Pipeline A must pump water into Pool Two at time *t1*; Pipeline B and C must pump water into Tower X and Y at time *t2*, respectively.

Table 1. The values of QoS parameters of the water treatment plant example

TU	MPC	BL	CPU usage	WTV
C11	10	20	(20,23)	(5,8)
C12	15	14	(10,12)	(10,12)
C13	13	17	(15,18)	(10,12)
C14	15	22	(5,7)	(8,10)
C21	16	28	(10,15)	(5,9)
C22	18	33	(15,18)	(4,7)
C23	20	20	(20,22)	(7,10)

Constraint (a) is a restriction of the design decision. Constraints (b) and (c) are the strict and orthogonal static QoS parameters. Constraints (d) and (e) are the dynamic QoS parameters. Constraint (f) is the time constraint. Table 1 includes all of the values of the QoS parameters requested from the knowledge base. Column 1 shows the identity of each treatment unit (TU), column 2 contains the maximum flow processing capacity (MPC) of each TU (million gallons/day), column 3 shows the current battery life (BL) of each unit (voltage), column 4 is the CPU usage of each TU (%), and the last column contains the water treatment volume (WTV) of each TU (million gallons/day). Figure 5 shows the Petri Net model of the project under constraints (a) and (f). The bars (i.e., transitions) at the same level of *t0, t1* and *t2* horizontally have the mechanism of the timing control.

QoS-UniFrame generates a reachability tree of the project based on strict and orthogonal static QoS. During



Water pumped from the River
Choose 3 Components from 4 Choices in Pool One
4 Design Options Go through Pipeline A
Choose 2 Components from 3 Choices in Pool Two
Clean Water goes to Tower X and Tower Y
Time Line
t0
t1
t2
C11
C12
C13
C14
A
C21
C22
C23
X
Y

Figure 5. The example of the Petri Net model representing the water treatment plant.

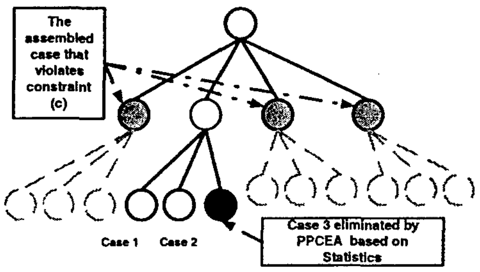
Table 2. The experimental results of the water treatment plant project

	Case 1	Case 2	Case 3
CPU Average	69.8223	73.9332	77.4793
CPU Worst	64.1087	75.0327	78.4904
WTV Average	40.7911	43.25	42.107
WTV Worst	36.2826	39.4127	37.1191
NO Best	11.8349	10.4933	11.215
NO Average	11.3491	10.1158	10.6731
NO Worst	9.483	8.4471	8.9652

the first level assurance, two *after* advice statements from Figure 3 are written and weaved into the source code of the tree construction. The first advice examines the satisfaction of the constraint (b), and the second advice assures the constraint (c). From the experimental result, QoS-UniFrame shows that C12 does not meet the constraint (c). Thus, only C11, C13, C14, C21, C22 and C23 will be chosen for pool one and pool two. At this stage, three assembled cases have survived: {C11,C13,C14,C21,C22}, {C11,C13,C14,C21,C23}, and {C11,C13,C14,C22,C23}. Subsequently, the CPU usage and water treatment volume (WTV) require the previous states and observations stored in the knowledge base. Table 1 contains the boundaries of the dynamic QoS requirements. At the second level, the parameter tuning approach written in PPCEA code is involved (Figure 4). First, two dynamic QoS constraints are examined independently by using addition. The predefined discard rate is 1.1, which means if the worst case is greater than 1.1 times this strict dynamic QoS requirement, the evaluated case is deleted. All of the predefined values of parameters needed for EAs are in Table 2. "Discard" is defined in section 3.4.3. "Maxgen" is the maximum number of generations (100) an EA can run. "Popsize" is the size of a population (value 100), "Pxover" is the crossover rate (0.5), and "Pmutation" is the mutation rate (0.7) [12]. Please note that, for brevity, only one parameter setting is represented in the paper. To obtain the best statistical results, a fitness function can be evaluated with various parameter settings in a programmable fashion during the execution of PPCEA code [11]. Table 2 contains the average results of each case after ten iterations at the second level. Case 1 represents {C11,C13,C14,C21,C22}, case 2 expresses {C11,C13,C14,C21,C23}, and case 3 is {C11,C13,C14,C22,C23}. "NO" stands for the non-orthogonal fitness function described below. Table 2 shows that {C11,C13,C14,C22,C23}'s average of ten worst cases is bigger than 1.1 times the constraint (d). Therefore, QoS-UniFrame tends to discard this design space. Case 3 does not meet the discarding policy, so QoS-UniFrame keeps its information for future use. Because CPU usage and water treatment volume are non-orthogonal dynamic QoS parameters, we defined a fitness function to address the mutual effect of CPU usage and water treatment volume. The fitness function is defined as below:

$$f(x) = (CPU\ Usage)/(Water\ Treatment\ Volume)$$

This function is treated as the statistical references for future investigation instead of a constraint. Finally, {C11,C13,C14,C21,C23} and {C11,C13,C14,C21,C22} are two survival cases statistically based on Figure 4.



The assembled case that violates constraint (c)
Case 1
Case 2
Case 3 eliminated by PPCEA based on Statistics

Figure 6. The Petri Net reachability tree of the water treatment plant example.

These experimental results show that QoS-UniFrame outperforms the OBDD approach [13] in the example of the water treatment plant project. At the first level, QoS-UniFrame cuts off 3 intermediate nodes, as shown in Figure 6. Each of these intermediate nodes have three more child nodes. Therefore, 9 more design spaces are eliminated before the end of reachability tree construction. The OBDD method, however, requires generating all 12 cases which is less efficient than QoS-UniFrame. In addition, by using the discarding policy at the second level, PPCEA statistically discards one more case. Therefore, QoS-UniFrame has better performance than the OBDD approach for this specific example.

5 Conclusion and the Future Work

The earlier that an error is detected in the software lifecycle, the less costly it is to fix [1]. QoS-UniFrame obeys this golden rule to reduce the design space at system assembly time. At the first level, the dynamic and parallel pruning approach is applied to expedite the pruning process. Only the feasible QoS systemic paths are generated by backtracking or branch-and-bound algorithms. At the second level, a fine-grained statistical approach is employed to further eliminate less probable QoS systemic paths. PPCEA also provides auxiliary statistical results as the reference at runtime. In addition, constructing Petri Net-based QoS modeling in the GME in collaboration with AspectJ facilitates customization, extensibility, flexibility, modularity and reusability. In conclusion, QoS-UniFrame provides a formal, manageable, scalable and semi-automatic approach to prune off unsatisfactory design spaces, and to validate a DRE system from its requirements at system assembly time. The design complexity of building DRE systems complying with numerous decisions, ordered components, events, and time can be further reduced than the OBDD method. For more details regarding QoS-UniFrame, please refer to [15].

QoS-UniFrame introduces a mathematical method (i.e., a fitness function) to solve the non-orthogonal QoS side effect problem. However, this approach is still not comprehensive and further research is necessary. For example, the priorities of the non-orthogonal QoS and the degree of the affectations among these QoS must be defined. Finally, QoS-UniFrame is a semi-automatic toolkit to explore, decrease and then assure the design spaces with constraints. System analysts would be required to have the basic knowledge of programming skills in AspectJ and PPCEA. A comprehensive automatic toolkit of design space exploration and assurance that eases system analysts and system engineers' workload is also the future direction of QoS-UniFrame.

References

[1] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[2] R. E. Bryant. Symbolic manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[3] L.A. Cortés, P. Eles, and Z. Peng. Formal coverification of embedded systems using model checking. In *Proc. 26th EUROMICRO Conf.*, pages 106–113, 2000.

[4] L.A. Cortés, P. Eles, and Z. Peng. Verification of embedded systems using a petri net based representation. In *Proc. 13th Intl. Symp. on System Synthesis*, pages 149–155, 2000.

[5] A. E. Eiben, R. Hintering, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Trans. on Evolutionary Computation*, 3(2):124–141, 1999.

[6] Z. Gu and K. G. Shin. An integrated approach to modeling and analysis of embedded real-time systems based on timed petri nets. In *Proc. 23rd Intl. Conf. on Distributed Computing Systems (ICDCS'03)*, pages 350–359, 2003.

[7] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms.* Computer Science Press, 1998.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Commun. of the ACM*, 44(10):59–65, 2001.

[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. European Conf. on Object-Oriented Programming (ECOOP'97), LNCS 1241*, pages 220–242, 1997.

[10] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, November 2001.

[11] S.-H. Liu, M. Mernik, and B. R. Bryant. Parameter control in evolutionary algorithms by domain-specific scripting language PPCEA. In *Proc. Intl. Conf. Bioinspired Optimization Methods and Their Applications (BIOMA'04)*, pages 41–50, 2004.

[12] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, 1996.

[13] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design space exploration and model synthesis. In *Proc. 3rd Intl. Conf. Embedded Software (EMSOFT'03), Springer-Verlag LNCS*, volume 2855, pages 290–305, 2003.

[14] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[15] QoS-UniFrame. http://www.cis.uab.edu/liush/QosUniFrame.htm.

[16] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, and C. C. Burt. A quality of service-based framework for creating distributed heterogeneous software components. *Concurrency and Computation: Practice and Experience*, 14(12):1009–1034, 2000.

[17] D. C. Schmidt. R&D advances in middleware for distributed real-time and embedded systems. *Communications of the ACM*, 45(12):43–48, 2002.

Quality of Service-Driven Requirements Analyses for Component Composition: A Two-Level Grammar++ Approach[1]

Shih-Hsi Liu[2], Fei Cao[2], Barrett R. Bryant[2], Jeff Gray[2], Rajeev R. Raje[3], Andrew M. Olson[3], and Mikhail Auguston[4]

Abstract

Component-based software engineering offers the opportunity to assemble entire systems from components. When applied to Distributed Real-Time and Embedded (DRE) systems, which components to assemble and how to assemble them are determined not only from functional correctness criteria but also assurance of the system's quality of service (QoS). This paper presents a grammatical QoS-driven approach to optimize component assembly by reducing the search space of assembly alternatives by eliminating infeasible components, with feasible components selected based on reasoning about non-functional requirements. The reasoning is realized by a rule engine with a knowledge base derived from the requirements phase of the software lifecycle. In addition, the grammatical approach introduces well-defined semantics among the components being composed. The semantics assist in precisely and efficiently evaluating the individual component QoS, as well as system-wide QoS in a programmable fashion. The result is to facilitate straightforward and manageable component composition analyses from the perspective of QoS requirements.

1 Introduction

Distributed Real-Time and Embedded (DRE) software systems are becoming increasingly complex. Such complexity can only be managed by Component-Based Software Engineering (CBSE), that is, building such systems from a collection of standardized and customized components. The integration of such components into a software system is the major effort in constructing such systems. Another dimension of such systems is the notion of Quality of Service (QoS), which transcends functional properties to include non-functional properties such as real-time and security issues. When DRE systems are constructed, QoS plays a critical role in determining the quality of the system. Along with functional specifications and models of the components, QoS attributes must also be specified and validated. The vision of the UniFrame project [9] is the development of techniques and tools that will enable software engineers to construct a DRE system by locating software components scattered about an organization or from third parties, evaluating the compatibility of heterogeneous components, generating connectors for the dissimilar pieces and validating a system composed from them.

This paper presents a grammatical QoS-driven approach to solve the challenges of black box component composition based on QoS. This approach expresses the system requirements in terms of QoS parameters and manipulates the QoS requirements using grammar rules which assure the correctness of the composition with respect to QoS and pre-conditions and post-conditions of each composition. This verification assists in eliminating the infeasible alternatives for any pre-condition or post-condition that does not satisfy the corresponding QoS constraints (i.e., facts) stored in the knowledge base. The knowledge base consists of specific composition rules for inferring the applicability of component composition. If all conditions are verified, the composition is assured. The systematic optimal solution of all QoS parameters can be evaluated by defining a specific QoS utility function of various QoS parameters. The specification of QoS requirements using grammar and rules facilitates the straightforward and manageable component composition analyses from the perspective of QoS parameters.

The paper is organized as follows: the next section provides background; section 3 proposes the concepts and an example; section 4 concludes the paper.

2 Background

The evolution of new techniques for software development is driven by the requirements of scalability within the growing complexity and size of modern software. To avoid developing scalable complex systems from scratch, CBSE enables the composition of commercial off-the-shelf (COTS) components, thereby benefitting software develop-

[1]This research is supported in part by U. S. Office of Naval Research award N00014-01-1-0746.

[2]Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, USA, {liush, caof, bryant, gray}@cis.uab.edu

[3]Department of Computer and Information Science, Indiana University-Purdue University-Indianapolis, Indianapolis, IN 46202-5132, USA {rraje, aloson}@cs.iupui.edu

[4]Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943-5193, USA {maugusto}@nps.navy.mil

ment by reusing and replacing components as needed. Software product lines [4] enrich the merits of CBSE by analyzing and constructing a set of software systems that share commonality and variability under specific considerations. The integration of CBSE and software product lines expedites the pace of software development, and proliferates the productivity of software products. The integration poses the following challenges for QoS-sensitive systems:

The Component Perspective Problem
Functional requirements define the functionality that systems should perform, and non-functional requirements specify constraints on system resources. Most systematic requirements analyses are component-driven [8], i.e., the analyses are based on the perspective of components and their functional requirements rather than non-functional requirements. The primary insufficiency of the component-driven analyses for QoS-sensitive system is that non-functional requirements are often tangled with functional ones. As numerous QoS characteristics require evaluation, separation of requirements concerns assists in manageably evaluating functional and non-functional requirements.

The Abundant Alternatives Problem
Hundreds of alternatives are generated based on the requirements of different composition decisions and permutations of selected components. The evaluation and management of abundant alternatives result in intensive workloads in the requirements phase.

The Composition Semantics Problem
Because component-driven analyses concentrate on the component units, the correlative composition semantics are not rich enough to state the composition influences on the QoS parameters. For example, the description of degradation and upgrade of certain QoS parameters is difficult by the component-driven composition semantics. Therefore, the evaluation of QoS parameters may not be performed in isolation, especially for some QoS parameters which mutually influence one another.

3 A Grammatical QoS-Driven Approach

Two-Level Grammar++ (TLG++) [3] is an object-oriented formal specification language, which consists of two Context-Free Grammars (CFGs) defining the set of parameters and the set of function definitions over the parameters, respectively. Originally, TLG++ was used for defining the syntax and semantics of programming languages: the first level consists of the production rules of the syntax and the second level interprets the semantics of these rules. TLG++ has been used for both specification of rules for component assembly [2] and for composing features to describe the characteristics of components [10]. In addition, TLG++ code can be automatically converted into Java using T-Clipse [7], an Integrated Development Environment for TLG++. In our approach, every QoS parameter is represented by a class of TLG++: the first CFG shows the components of alternatives and the necessary parameters used for the function definitions. The second CFG describes the function definitions, which include the reasoning operations and computational operations (i.e., composition semantics) regarding QoS parameters. The reasoning operations are used for analyzing and verifying pre-conditions and post-conditions of each composition. For the pre-conditions, preliminary queries verify that the components own the appropriate functions operating the QoS. Analytic queries then request the QoS information of specific components. For the post-conditions, the conclusive queries send back the composed "pattern" (i.e., the selected components and the QoS dataflow among these components) to avoid any conflict with respect to the constraints; namely, verification of post-conditions. If preliminary, analytic or conclusive queries return false, the alternative is infeasible and discarded.

We use Jess [5] as the underlying rule inference engine for reasoning about alternatives' feasibility regarding QoS requirements. Jess is a forward and backward chaining rule engine for the Java platform, which bridges Java and the rule-based language. Jess includes a Java library for defining rules, facts and queries, and for invoking the rule engine. The knowledge base accumulates the facts and rules regarding the components and QoS parameters. Queries request answers inferred from the facts and rules stored in the knowledge base. The querying results obtained from the rule engine are converted into interpretable Java objects for further processing tasks written in Java.

The primary concepts and motivations of applying TLG++ to a QoS requirements analyses approach in the context of CBSE and software product lines assume the following: (a) the components, having functions computing a QoS parameter, are like the operands of an expression; (b) composition semantics are treated as the operator of two (sets of) components; (c) production rules[5] are the counterparts of composition decisions, which imply the dataflow of the QoS parameters among components. Constructing a system is actually the same as defining a programming language with syntax and semantics. Under such a concept, Extended Backus-Naur Form (EBNF) [1] can represent mandatory, alternative (i.e., one of), optional and "OR" (i.e., more of) features of components involved in a software product line, as in Feature-Oriented Domain Analysis [6]. The syntax trees generated by applying different sets of production rules can be treated as the counterparts of the alternatives of a software product line. TLG++, consisting of two tightly coupled CFGs, is appropriate for the grammatical QoS-driven approach to define customized and comprehensive semantics for component composition.

[5]Production rules may have ambiguity, left recursion and left factoring problems. Analyzers should avoid these grammatical problems.

Figure 1 shows the procedures for analyzing systematic QoS requirements. First, analyzers write all QoS parameter classes in TLG++, which define the involved components and the composition semantics among the components regarding the QoS parameter. T-Clipse transforms TLG++ into Java. Second, the strict QoS parameters are evaluated, because they are the strict feasibility criteria for the alternatives. Third, all orthogonal QoS parameters are individually evaluated, and every set of non-orthogonal QoS is collectively estimated. Orthogonal QoS parameters imply that adaptation will not influence other QoS parameters, yet non-orthogonal QoS parameters substantially influence other QoS parameters. After all sets of non-orthogonal QoS are assured, the cumulative goals, the final selection criteria of alternatives, can be computed by a user-defined algebraic function over all assured QoS parameters. All of the fulfilling patterns of the software product lines will be stored in the knowledge base for the future queries. In the situations that strict, orthogonal or non-orthogonal QoS are not satisfied, a new (set of) component(s) will be selected as a new alternative to be evaluated.

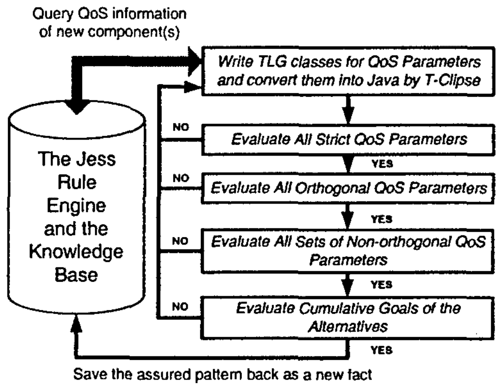

Query QoS information of new component(s)
Write TLG classes for QoS Parameters and convert them into Java by T-Clipse
The Jess Rule Engine and the Knowledge Base
NO
Evaluate All Strict QoS Parameters
YES
NO
Evaluate All Orthogonal QoS Parameters
YES
NO
Evaluate All Sets of Non-orthogonal QoS Parameters
YES
NO
Evaluate Cumulative Goals of the Alternatives
YES
Save the assured pattern back as a new fact

Figure 1. The procedures of the approach.

Figure 2 shows the user-defined grammars for each QoS parameter: the C_i are the terminals that represent components, and D_j, E_k, and F_l are nonterminals that describe the composition decision and the QoS dataflow. The left box, the middle box, and the right box, are the grammars for *Security, Signal*, and *a set of non-orthogonal QoS (Time, CPU Usage,* and *Battery Life)*, respectively. Please note that some production rules have left factoring, which may be eliminated as described in [1].

1 Security → C1 C2 D1	1 Signal → C1 C2 E1	1 CPU → C1 F1 | C2 F2
2 D1 → C3 D2 | C4 D3	2 E1 → C3 E2 | C4 E3	2 F1 → C2 C4 F3 | C3 C4 F4
3 D2 → C4 C5 | C5 C6	| C5 E4	3 F2 → C5 C6 F5 | C5 | C6 F6
4 D3 → C5 D4 | C5 C7	3 E2 → C6 C7	4 F3 → C7 C6
5 D4 → C3 C7	4 E3 → C3 C5 E5 | C3 C6	5 F4 → C2 C5
	5 E4 → C4 C6 C7	6 F5 → C3 C7
	6 E5 → C7	7 F6 → C1 C4

Figure 2. Grammars for QoS parameters

The cascading scenario is introduced to evaluate orthogonal QoS parameters. A set of components is chosen as the starting point of a QoS dataflow. The consequent components are opted by specific decisions such as *AND* and *OR*. *AND* means the dataflow streams into a set of components, and *OR* implies the new alternatives of the software product line are generated. As a QoS dataflow requires a new composition decision, a new TLG++ class is written: the parameters include the new components being selected, and the functions define the composition semantics between its ascendant and itself with respect to the QoS dataflow.

The upper box of Figure 3 represents the TLG++ class for the first production rule in Figure 2, the starting point of the *Security* QoS dataflow. In the upper box, line 2 comprises the first CFG that defines the selected components for the second CFG. Lines 3 to 29 comprise the second CFG that describes the semantics for composition, including computational and reasoning operations. Lines 3 and 4 verify the pre-conditions of Components Comp_1 and Comp_2. In "queryComponent" (lines 12 to 27), the functions of the Java API for the Jess rule engine (e.g., executeCommand) are invoked. Lines 13 to 15 define the query for searching the facts of QoS parameters. Lines 18 to 21 define where the querying results should be stored. Lines 23 to 25 comprise the semantics that define how to fetch the elements of the facts of the QoS parameter. After verifying the pre-conditions, lines 5 to 6 compute the QoS value based on the composition semantics defined in line 28. Finally, line 9 verifies the post-condition of the composition by checking if the composed QoS value is out of range. The lower box of Figure 3 is the Security_2 class for composing Security_1 using the second production rule based on the cascading scenario. In the lower box, the semicolon in line 2 means there are optional components for the software product line (i.e., the counterpart of "|" in the EBNF). Therefore, this box contains two composition semantics for components Comp_3 and Comp_4, respectively. *Signal* is defined in the similar way using its grammar in Figure 2.

For non-orthogonal QoS analyses, it is difficult to find the optimal balance when one non-orthogonal QoS parameter increases and the other one decreases. The coarse-grained scenario extends the cascading scenario for non-orthogonal analyses. All sets of non-orthogonal QoS parameters are written in TLG++ classes using the cascading scenario. A TLG++ class defines a weighted algebra function over each set of non-orthogonal QoS parameters (in this paper, *Time, CPU Usage,* and *Battery Life*) to discover the maximum value. Figure 4 shows the decision trees of five QoS parameters, expressing every composition decision as a branch of the tree. If any component in a QoS dataflow violates strict QoS (i.e., gray nodes), the following nodes (i.e., stripe nodes) are eliminated. The cumulative goal is computed by a user-defined algebraic function over all feasible goals of QoS parameters.

```
class Security_1 implements Serializable
Product_Line :: Comp_1 Comp_2. // All other parameter declarations ignored
Query_1 := semantics of queryComponent with Comp_1://verify pre-cond. of Comp_1
Query_2 := semantics of queryComponent with Comp_2;//verify pre-cond. of Comp_2
Query_3 := if Query_1 && Query_2, then semantics of minimum with
 Comp_1 and Comp_2, else False, end if:
//if both Query_1 and Query_2 are true, compute the composition semantics of
//Comp_1 and Comp_2. Otherwise, stop analyzing the alternative
Query_4 := semantics of queryPattern with QoSValue;//verifies post-cond. check range
if Query_4, then MyRete semantics of UpdatePattern, else "False", end if.
//if Query_4 is true, the composed pattern is assured. Update the pattern to KB
semantics of queryComponent with Component :
  MyRete semantics of executeCommand with "(defquery QoSSearch (declare
  (variables ?comp)) (qos (mycomponent ?comp) (myfunc ?func) (qoslow ?low)
    (qosup ?up)))":
  //define the Jess query for the QoS parameter, which has the fields of components,
  //functions, lower bound and upper bound.
  ValueVector_1 := ValueVector semantics of addAll withValue_1:
  //Store the fields into ValueVector, an API provided by Jess' Java library
  MyRete semantics of store with "RESULT" and MyRete semantics of RunQuery
    with "QoSSearch" and ValueVector_1;
  //Store the result of the query into the RESULT variable
  MyRete semantics of executeCommand with "(run-query QoSSearch "+
    component+")'"; //Run the query component is the variable of the query
  Iterator_1 := MyRete semantics of fetch with "RESULT";//RESULT saved to Iterator
  if Iterator_1 != null, then return TRUE, else return FALSE, end if.
  //if the first field has no component defined, the pre-condition is not verified
semantics of minimum with Component1 and Component2 ://...ignored
//semantics of queryPattern, and UpdatePattern are ignored.
end class
```

```
class Security_2 implements Serializable. // All other parameter declarations ignored
Product_Line :: Comp_3 ; Comp_4. //Comp_3 OR Comp_4 as alternatives
semantics of ProductLine_1 with Component1 : //semantics for Comp_3 OR Comp_4
  Query_1 := semantics of queryComponent with Component1://verify pre-cond.
  //queryComponent has same semantics in Figure 3
  if Query_1, then semantics of addition with Security_1 and Component1.
  else False, end if:
  Query_2 := Rete semantics of queryPattern with QoSValue;
   if Query_2, then Rete semantics of UpdateFact, Rete semantics of UpdatePattern, else
   "Composition False", end if. //verify the post-condition
semantics of addition with Component1 and Component2 : //...ignored
//semantics of queryPattern, UpDateFact and UpdatePattern are ignored here.
end class
```

Figure 3. Security_1 and Security_2 in TLG++

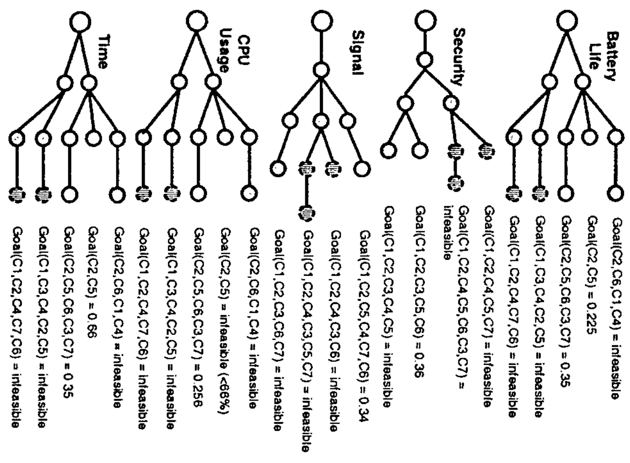

Battery Life
Goal(C2,C6,C1,C4) = infeasible
Goal(C2,C5) = 0.225
Goal(C2,C5,C6,C3,C7) = 0.35
Goal(C1,C3,C4,C2,C5) = infeasible
Goal(C1,C2,C4,C7,C6) = infeasible
Security
Goal(C1,C2,C4,C5,C7) = infeasible
Goal(C1,C2,C4,C5,C6,C3,C7) = infeasible
Goal(C1,C2,C3,C5,C6) = 0.36
Goal(C1,C2,C3,C4,C5) = infeasible
Signal
Goal(C1,C2,C5,C4,C7,C6) = 0.34
Goal(C1,C2,C4,C3,C6) = infeasible
Goal(C1,C2,C4,C3,C5,C7) = infeasible
Goal(C1,C2,C3,C6,C7) = infeasible
CPU Usage
Goal(C2,C6,C1,C4) = infeasible
Goal(C2,C5) = infeasible (<66%)
Goal(C2,C5,C6,C3,C7) = 0.256
Goal(C1,C3,C4,C2,C5) = infeasible
Goal(C1,C2,C4,C7,C6) = infeasible
Time
Goal(C2,C6,C1,C4) = infeasible
Goal(C2,C5) = 0.66
Goal(C2,C5,C6,C3,C7) = 0.35
Goal(C1,C3,C4,C2,C5) = infeasible
Goal(C1,C2,C4,C7,C6) = infeasible

Figure 4. Decision trees of QoS parameters

4 Conclusions

The grammatical QoS-driven approach defines the syntax of software product lines, and the semantics of the component composition from the QoS parameter perspective. The approach eases the burden of management and evaluation of QoS that the component-driven approaches suffer from. It also achieves three goals: reducing the infeasible alternatives, assuring the feasible ones, and manageably evaluating orthogonal QoS and mutually-influenced QoS. Finally, a stand-alone inference engine separates the inference concern for component composition.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] B. R. Bryant, M. Auguston, R. R. Raje, C. C. Burt, and A. M. Olson. Formal specification of generative component assembly using Two-Level Grammar. In *Proc. of 14th Intl. Conf. on Software Engineering and Knowledge Engineering*, pages 209–212, 2002.

[3] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an object-oriented requirements specification language. In *Proc. of the 35th Hawaii Intl. Conf. on System Sciences*, 2002. http://www.hicss.hawaii.edu/HICSS_35/HICSS papers/PDFdocuments/STDSL01.pdf.

[4] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[5] E. J. Friedman-Hill. *Jess 7.0, The Rule Engine for the Java Platform*. Sandia National Laboratories, 2005.

[6] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[7] B.-S. Lee, X. Wu, F. Cao, S.-H. Liu, W. Zhao, C. Yang, B. R. Bryant, and J. G. Gray. T-Clipse: An integrated development environment for Two-Level Grammar. In *The OOPSLA'03 Eclipse Technology Exchange Workshop*, pages 91–95, 2003.

[8] M. Matinlassi. Comparison of software product line architecture design methods: COPA, FAST, FORM, KorbA and QADA. In *Proc. of the 26th Intl. Conf. Software Engineering*, pages 127–136, 2004.

[9] R. R. Raje, B. R. Bryant, M. Auguston, A. M. Olson, and C. C. Burt. A QoS-based framework for creating distributed and heterogeneous software components. *Concurrency and Computation: Practice and Experience*, 14:1009–1034, 2002.

[10] W. Zhao, B. R. Bryant, F. Cao, R. R. Raje, M. Auguston, C. C. Burt, and A. M. Olson. Grammatically interpreting feature composition. In *Proc. of 16th Intl. Conf. on Software Engineering and Knowledge Engineering*, pages 185–191, 2004.

Marshaling and Unmarshaling Models Using the Entity-Relationship Model*

Fei Cao, Barrett R. Bryant,
Wei Zhao, Carol C. Burt
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Boulevard, Birmingham, AL 35294, USA
{caof, bryant, zhaow, cburt} @cis.uab.edu

Rajeev R. Raje, Andrew M. Olson
Department of Computer and Information Science
Indiana University-Purdue University-Indianapolis
723 W. Michigan Street SL 280, Indianapolis, IN 46202, USA
{rraje, aolson}@cs.iupui.edu

Mikhail Auguston
Computer Science Department
Naval Postgraduate School
1 University Circle, Monterey, CA 93943, USA
maugusto@nps.edu

ABSTRACT
Software systems are usually designed and documented with the aid of visual modeling notations. Visual modeling notations keep evolving over the years in tandem with visual modeling tools, and the tight binding in between impedes the exchanging of modeling assets, which causes a spatial isolation of the models. Another problem with legacy software models is that they are isolated temporally in the early phases of the software engineering life cycle without reaching out to the later phases. This paper presents an approach for breaking both spatial and temporal isolation of software models by marshaling and unmarshaling models using the Entity-Relationship (ER) model, thus providing a promising way for evolving model-driven software development.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords
Marshaling and unmarshaling models, Modeling and meta-modeling, Entity-Relationship model

1. INTRODUCTION
Software systems are usually designed and documented with the aid of visual modeling notations. Visual modeling notations keep evolving over the years in tandem with visual modeling tools, and the tight binding in between impedes the exchanging of modeling assets. Above all UML[1] stands out as the *de facto* standard modeling language. But other non-UML based modeling notations abound as evidenced in such publications as JVLC[2]. Meanwhile, a lot of work has been done to converge the diagram notations in the new version of modeling notations, as is mentioned in the recent interview with Keith Short[3]. But to converge all the legacy software modeling assets by reengineering into new generation notations and totally discarding old legacy modeling notations is not only time-consuming, but also not cost-effective. Depending on different usage scenarios, there is a need for marshaling models across different modeling facilities to take advantages of the leverages provided by existent modeling facilities.

The term *Marshaling* comes from the distributed computing area where heterogenous data types are always translated into some common data type over the network so as to be consumed at the other end of the distributed environment, where the common data type is *unmarshaled* again into another environment-specific data type. Here we use the ER model [2] to represent the "common data type", i.e., the intermediate model when exchanging and evolving models. The rationales are as follows:

- **Sufficiency.** Even though UML is widely adopted in software modeling, which seems to justify the use of UML as a common model for exchanging model assets across modeling facilities, UML is not convenient for model serialization, thus not fit for modeling asset exchange and evolution. In fact, the object diagram [1], for which UML is used to capture and store the snapshot of software system state, is represented virtually in an Entity (object) and Relationship (links) model. Moreover, the UML modeling language has its roots in the ER model, and the latter is already widely used as the foundation for CASE tools in software engineering and repository systems in databases[4].
- **Necessity.** Not only models, but also meta-models are in need of exchanging and evolution; the justification for the latter is obviously the same as the former. Therefore, the intermediate model should be

* This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

[1] Unified Modeling Language-http://www.omg.org/uml
[2] Journal of Visual Languages and Computing-http://www.elsevier.com/locate/jvlc
[3] Interview with Keith Short, http://www.theserverside.net/talks/library.tss#KeithShort.
[4] http://bit.csc.lsu.edu/~chen/chen.html

expressive enough to be at the meta-meta model level in the meta-level stack [3]. The meta-meta-model is described by the Meta Object Facility (MOF)[5], which is a set of constructs used to define meta-models. The MOF constructs are the MOF class, the MOF attributes and the MOF association. These constructs correspond to an ER representation (by using an Entity to represent a MOF class), which indicates that the ER representation is semantically equivalent to MOF fundamentally. Therefore, we believe the ER representation is the right vehicle to play the *dual roles* of marshaling both models and meta-models to break the spatial isolation of software models. Also, other non-UML based languages, even though not as popular, are abundantly present, for which UML is not an omnipotent cure.

Recent years have seen the emergence of the Model Integrated Computing (MIC) [7] paradigm, which moves a step further to break the isolation of models from implementation and the subsequent phases in the software engineering life cycle. In MIC, a meta-model is created to define a model construction language, and a generator is also to be created based on the meta-model to synthesize the constructed models by traversing the model tree. In this way, a model can be more accurately interpreted for code generation than the direct mapping-based approach such as using profiler or stereotype in Rational Rose [3]. Toward that end, this paper presents an approach for marshaling software models to ER models, which, by taking advantage of the dual roles of ER models, are unmarshaled into an environment-specific meta-model to be integrated into MIC. Consequently, not only the spatial isolation, but also the temporal isolation of software models can be broken.

This paper is organized as follows: Section 2 briefly provides an overall picture of this approach. Section 3 uses Web Services (WS) [5] modeling as a proof-of-concept example to illustrate the whole process. Section 4 describes the related work. We conclude in section 5 with a brief description of future work included.

2. OVERVIEW

Figure 1 shows the process of marshaling and unmarshaling models. Generic Modeling Environment (GME) [4] is the tool for MIC paradigm, and we use it as the targeted tool environment for describing destination meta-models, whereupon the domain-specific modeling environment can be constructed. Through the process flow as is directed by the arrows, meta-models can be elicited from models with an automatable process as opposed to traditional practice, for which the meta-model is constructed in an error-prone, ad-hoc way. Consequently, models of legacy systems can be evolved toward the MIC paradigm for model-driven software development.

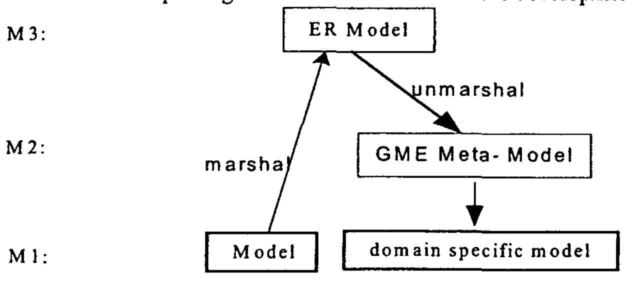


Figure 1. Marshaling and unmarshaling models

[5] Meta-Object Facility - http://www.omg.org/technology/documents/formal/mof.htm

3. THE APPROACH

3.1 A Web Services Modeling Example

Modeling Web Services (WS) is a promising way for service description and orchestration at a higher level. As the scope of this paper is about marshaling and unmarshaling models, the elicitation of models from requirements is skipped here.

One of the characteristics of a meta-model is that it treats not only the models, but also the inter-relationships among models as first-class entities. We derive meta-models by abstracting models and their inter-relationships. Therefore, for the models, even though they are represented as UML diagrams here as the starting point of the marshaling/unmarshaling process, they will not compromise the generality of the approach as is described in the remainder of the paper. To be specific, our approach of marshaling and unmarshaling WS models consists of two steps:

1) Marshal models by converting the OO class diagram to an ER-based meta-model, for which the *relationship* corresponds to *aggregation*, *association*, *generalization*, and *dependency*, while the *entity* corresponds to *class*.
2) Unmarshal models by mapping the ER-based meta-model to the tool-specific (here GME in particular) meta-model to create a WS modeling environment.

The UML class diagram of WSDL elements is shown in Figure 2.

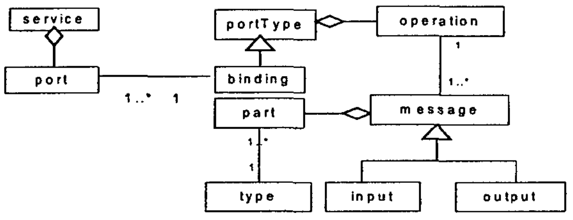


Figure 2. The architecture of WS description elements

The WS ***messages***, which are either ***input*** or ***output*** messages, are composed of ***parts***, each of which corresponds to a specific data ***type***. The ***portType*** is an abstract WS interface definition, where each contained element, i.e., the ***operation***, defines an abstract method signature. The operation uses messages as its parameters. ***Binding*** represents an instantiation to the abstract ***portType*** with concrete protocol and data type. ***Service*** is a collection of ***ports***, denoting a deployment of a binding at a specific network location.

3.2 Marshaling the WSDL Model

Figure 3 gives the meta-model of WSDL in ER form (without considering the extension part enclosed with the dashed lines), which is derived by representing the links (*association*, *generalization*, *dependency*) in the class diagram in Figure 2 as a *relationship* in Figure 3, as well as representing those classes as an *entity* accordingly. Note we ignore *type* in the meta-model of Figure 3, because we can put type directly as the attribute of the part element. Also note we will not annotate the attributes to the entities and relationships in the ER representation as the focus here is about the model marshaling and unmarshaling; the attributes will be annotated in the GME meta-model as shown later.

When modeling WSDL for real business domain services implemented with specific technology, we use the *generalization* relationship to extend those WSDL elements in Figure 3 rather than embedding the business domain service information as attributes to those WSDL elements. This avoids obfuscation of business and

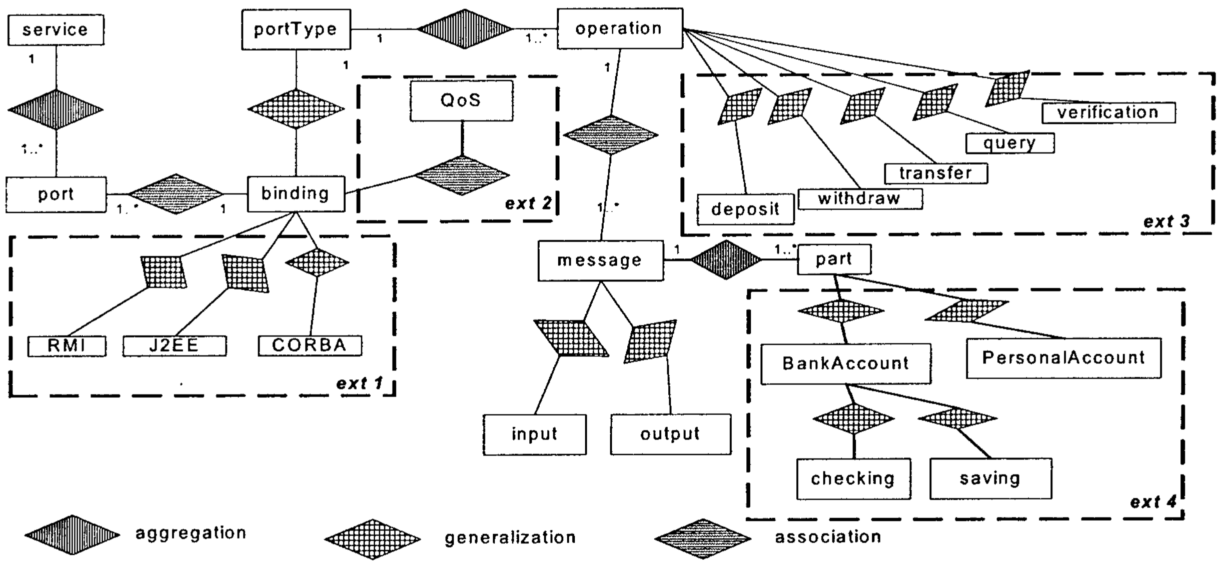


Figure 3. The ER-based meta-model of banking Service WSDL: the three parts enclosed with dashed line represent the extended part to the WSDL meta-model.

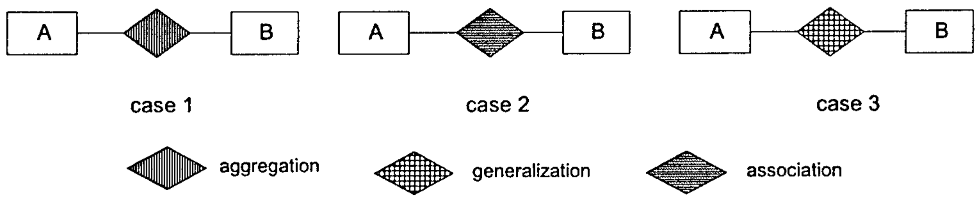


Figure 4. The cases of mapping from ER-based Meta-model to GME-based meta-model based on the relationship in ER representation.

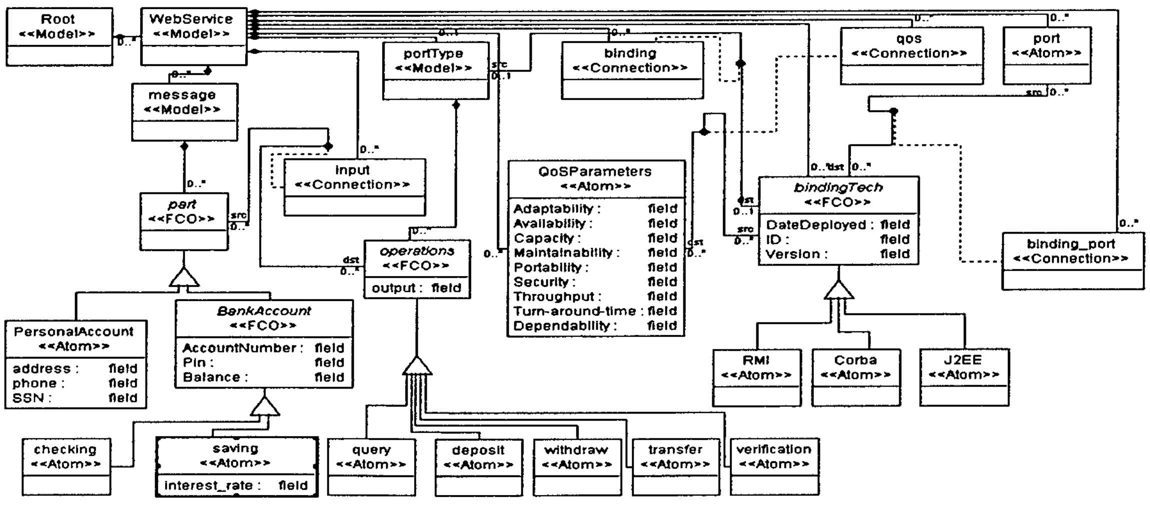


Figure 5. The meta-model of banking domain WSDL in GME

technology domain structure (meta-models of business/technology domain applications) with WSDL elements, and provides a separation of concerns toward domain-specific model refinement. The business domain information applies a generalization relationship to the operation entity, and technology domain information applies a generalization relationship to the binding entity. To exemplify, below is a simple banking domain service specification:

```
  A bank provides the service for users to
set up accounts.  Account information includes
personal data including Name, SSN, phone
number, address, and account data including
Account Number, PIN, Transaction Record,
Balance.  There are two types of accounts:
checking account and savings account.
  For the bank side, it provides such
services as: Account Verification, Account
Query, Deposit, Withdraw, and Transfer.
  The banking service implementation may use
such technology as RMI, J2EE, and CORBA. Also
it will enforce some Quality of Service (QoS)
requirements such as Availability,
Dependability, Capacity.
```

Figure 3 shows the ER-based meta-model of this banking service WSDL (including those parts enclosed by dashed line). The elicitation of models from natural language requirements is beyond the scope of this paper. As can be seen from the figure, a typical business domain service represented as WSDL involves the extension of ER elements, which is associated to almost all the elements of WSDL. Nevertheless, by using the ER-based meta-model, such extension still keeps the original WSDL meta-model as shown in Figure 3 without being restructured.

3.3 Unmarshaling the WSDL Model

In GME, the containment relationship is represented by using a *model* element (tagged with <<*model*>>), which, in contrast to an *atom* element (tagged with <<*atom*>>), can contain other modeling elements. Also the contained elements can be promoted as *ports* of the model to have direct connections with external modeling elements. GME uses a *root model* as an entry point of access to all the modeling elements. Also, the *relationship* of ER is represented in GME as a first-class modeling element, *connection* (tagged with <<*connection*>>), with a *connector* in the form of a dot to associate this relationship with two modeling elements (entities).

The mapping from the ER-based meta-model to the counterpart in GME is based on the relationships in the ER representation. Three cases are involved as is shown in Figure 4:

1) A contains B

In this case, A can be modeled as a *model* element in GME containing B.

2) B is associated to A

In this case, a *connection* can be added to be associated with the A and B representations in GME. The connection element can be named with respect to A's or B's properties as a kind of tag, e.g., the tag can be named as the combination of both A's name and B's name. Note when the situation as described in case 3 applies, then this tag should be named as in case 3.

3) B is specialized from A

In this case, A is rendered by an abstract FCO (First Class Object, tagged with <<*FCO*>>, represents an abstract generalization of other modeling constructs), a modeling element to be used as an abstract interface in GME, and B is represented as an inherited class to that FCO. Note there are two special treatments here: firstly, for the input/output elements of Figure 3, they are only used to tag the *connection* (named either "input" or "output") between message entities and its interconnecting entities in GME; secondly, the generalization relationship between binding and portType is actually treated as an association when modeling in GME, because the binding entity actually attaches values of the chosen protocol to the portType in WSDL rather than in the real sense of inheritance.

Figure 5 shows the meta-model created by mapping from the WSDL meta-model of the banking domain with ER representation to that in the GME strictly observing the above mapping rules. The model WebService corresponds to the service entity in Figure 3. The boxed part of the models in Figure 5 are attributes for the related models to be instantiated in the modeling phase, described in the next section.

3.4 The Domain Specific Modeling Environment

After a meta-model is derived by marshaling and unmarshaling models, a domain specific modeling environment (which is also a crucial part of MIC) can be created based upon the meta-model. To complete the description of the model evolution process shown in Figure 1, Figure 6 shows the screenshot of the banking-domain WS modeling environment based on the meta-model illustrated in Figure 5. The lower-left corner provides the modeling elements that can be dragged and dropped in the upper-left pane for constructing a banking service model. The names of the models in the lower-left pane represent the meta-model names (*kind names*); when those models are dragged to the above pane, the model name can be changed to reflect the meaning of the model in the domain-specific context, which we call a *context name*. Furthermore, the domain-specific model can be traversed and interpreted in terms of code generation using the GME Builder Object Network (BON) framework [4].

4. RELATED WORK

The ER model, because of its powerful modeling capacity, can be used as an intermediate form for model-to-model and meta-model-to-meta-model exchange. Because of the dual role that the ER model can play, it is treated as an intermediate form for model-to-meta-model elicitation, which is the theme of this paper. This idea is very similar to grammar inference [6], where a grammar can be inferred from language examples. But the two approaches are applied at different abstraction levels. XMI[6] provides a standard mapping from MOF-based nodels to XML, which can be exchanged between software applications and tools. In comparison, ER-based model marshaling and unmarshaling represents a design-level approach for evolving design assets, without being restricted to low-level data representation specifics. Also, note that the XMI-based approach uses top-down mapping, while the ER-based approach uses bottom-up mapping as is illustrated in Figure 1, which offers a means for meta-model recovery for evolving legacy software models into Model Integrated Computing.

[6] XML Metadata Interchange - http://www.omg.org/technology/documents/formal/xmi.htm

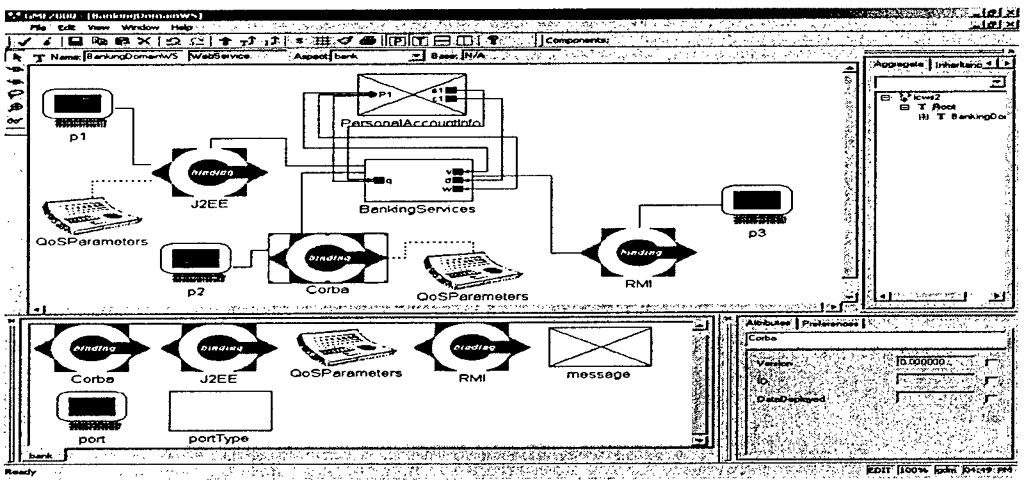


Figure 6. The banking domain-specific WS modeling environment.

Model Driven Architecture (MDA)[7] is about mapping Platform Independent Models (PIM) to Platform Specific Models (PSM) for engineering legacy software systems so as to be integrated into new platform. However, the core part of mapping technology for MDA is either ad-hoc or pre-mature before MDA can be fully adopted in industry. ER-based model marshaling and unmarshaling offers a potential solution to address this problem systematically. It has been observed that ER representation has been adopted in defining Knowledge Discovery Meta-Model (KDM)[8] and Ontology Definition Meta-Model (ODM)[9] in OMG, which underscores the role that ER plays for model marshaling and unmarhaling.

5. CONCLUSION AND FUTURE WORK

Legacy software models are widely existent and heterogeneous in their own graph syntax, and there are two types of isolation in its application: Spatially, models are isolated from being exchangeable over software applications and tools; Temporally, models are isolated in the early phases of the software engineering life cycle. These two types of isolation status of software models restrict their usability and capacity. Toward that end, a model marshaling and unmarshaling approach is presented based on the ER model, a simple, yet powerful modeling notation. This approach offers a promising way to break not only spatial isolations, but also temporal isolation by evolving legacy software models toward MIC for fully exploiting models throughout the software engineering life cycle. In particular, this paper uses a WS modeling example to illustrate an automatable process on how legacy software models can be migrated toward a MIC-oriented environment.

To ultimately automate the marshalling and unmarshaling process, future work will involve representing various models as well as ER models in the form of proper XML specifications, whereupon the automation process can be applied by XML transformation technology such as XSLT[10]. The ER model is easy to be represented in XML because of its simple structure. An Eclipse-based ER modeling tool such as [8] that can generate XML specifications from ER models will be helpful in this regard. The models in GME can be exported and imported as XML. Therefore, an XML specification for an ER-model can be directly transformed to the expected XML specification for destination meta-models and loaded into GME consequently. Note that the simple structure of ER models does not require an XMI–based data representation. Moreover, such existent tool as GME does not use XMI for model serialization and deserialization, for which a simpler and more flexible XML schema is desired for marshaling and unmarshaling models.

6. REFERENCES

[1] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[2] P. P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Trans. Database Systems*, 1(1), 1976, 9-36.

[3] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[4] *GME 2000 User's Manual, Version 2.0.* ISIS, Vanderbilt University, 2001.

[5] S. Graham, S. Simeonov, T. Boubez, D. Davis, G. Daniels, Y. Nakamura, R. Neyam. *Building Web Services with Java.* SAMS, 2002.

[6] C. de la Higuera. Current Trends in Grammatical Inference. In *Proc. Joint IAPR Int. Workshops SSPR & SPR 2000*, 2001, 28-31.

[7] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, 34(11), 2001, 44-51.

[8] S. Zhou, C. Xu, H. Wu, J. Zhang, Y. Lin, J. Wang, J. Gray, B. R. Bryant. E-R Modeler: A Database Modeling Toolkit for Eclipse. In *Proc. 42th ACM Southeast Conf.*, 2004, 160-165.

[7] http://www.omg.org/mda/
[8] http://www.omg.org/cgi-bin/doc?lt/2003-11-4
[9] http://codip.grci.com/odm/draft/submission_text/ODMPrelimSubAug04R1.pdf

[10] http://www.w3.org/TR/xslt

A Meta-Modeling Approach to Web Services

Fei Cao, Barrett R. Bryant, Wei Zhao, Carol C. Burt
University of Alabama at Birmingham
{caof, bryant, zhaow, cburt} @cis.uab.edu

Rajeev R. Raje, Andrew M. Olson
Indiana University-Purdue University-Indianapolis
{rraje, aolson}@cs.iupui.edu

Mikhail Auguston
Naval Postgraduate School
auguston@cs.nps.navy.mil

Abstract

Web Services (WS) technology is becoming pervasive in the development of distributed systems and is an appealing vehicle for service presentation and horizontal integration. On the other hand, Model Integrated Computing (MIC) offers a means of system integration in the vertical direction by using domain-specific modeling, and then synthesizing the software system from the high-level model using a model-specific generator. This paper presents a meta-modeling approach to WS to explore the application of MIC in WS development and its contribution.

1. Introduction

Web Services (WS) technology emerges as a Service Oriented Computing (SOC) ([8], [9]) paradigm to provide a platform-independent solution for system integration *horizontally*: WS is built upon open standard XML and HTML for service description and transportation, and software systems can be presented as WS so as to be exported and consumed by heterogeneous peers in the distributed environment.

For service description in Web Services Description Language (WSDL), though its XML-based representation is easy for machine processing using widely existent XML parsers, such specification is not straightforward for human comprehension, with service architecture lost in the pure textual form, and hand-crafting service description with WSDL is error-prone. To overcome this problem, there are tools on the horizon such as AXIS[1], and the Microsoft .Net framework that provide the capacity of automatically generating WSDL by parsing implementation code (such as Java and C#), and vice versa. However, WSDL represents the design level knowledge, and the process of generating WSDL from implementation is in contradiction to the general practice of software engineering, for which the design phase precedes the implementation phase. We believe generating WSDL from the design-level model directly offers an appropriate solution.

In this paper we present a meta-modeling approach to WS based on the principles of Model Integrated Computing (MIC) [5]. In MIC, meta-models can be used to define modeling language. Consequently, WS artifacts (WSDL) can be automatically generated from the WS model with generators. The Generic Modeling Environment (GME) [4] is a tool realizing MIC for creation of domain-specific models.

As is shown above, meta-modeling constitutes the corner stone In MIC. This paper is not intended to demonstrate the meta-modeling approach to all aspects of WS (e.g., discovery and orchestration) exhaustively, which is not possible because of the space limitation, but rather to focus on the elicitation of a tool-independent meta-model from WS requirements specifications, and then to map the too-independent meta-model to a tool-specific meta-model (here GME meta-model in particular), which is to be used throughout the main phases of MIC. This paper is organized as follows: Section 2 details the meta-modeling approach. Section 3 describes the related work. Section 4 draws the conclusion.

2. Meta-Modeling of WS

2.1 Meta-modeling WSDL using Entity Relationship (ER) representation

Object-Oriented class diagrams are often used to document software system architecture. The architecture of WSDL elements can be described as in Figure 1. Figure 2 gives the meta-model of WSDL (by removing the extension part enclosed with a dashed line), which is derived by representing the links (*association, generalization, dependency*) in the class

[1] http://ws.apache.org/axis/

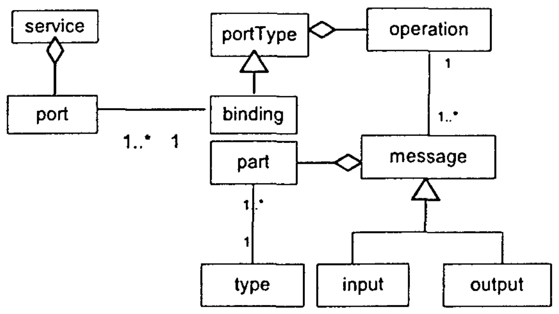


Figure 1: the Architecture of WS Description Elements

diagram in Figure 1 as a *relationship* in Figure 2, as well as representing those classes as an *entity* accordingly.

Note we ignore *type* in the meta-model of Figure 2, because we can put type directly as the attribute of the part element. Also note we will not annotate the attributes to the entities and relationships in the ER representation as the focus here is about the meta-model evolution; the attributes will be annotated in the GME meta-model as shown later.

The meta-model is represented by the ER representation [2] rather than by UML. The justification of using ER representation as an intermediate meta-model is as follows:

Different meta-modeling tools such as GME may adopt its own meta-model paradigm. Thus, there is a need for a tool-independent meta-model representation as an intermediate form for meta-model transformation, so that tool-dependent meta-models can be evolved into each other and used across different meta-modeling tools. This intermediate meta-model representation should be generic enough to describe a meta-meta-model, which resides at the top level (M3 level) of Model Driven Architecture (MDA)[2] metalevel stack [3]. The meta-meta-model used to define UML meta-models is described by the Meta Object Facility (MOF)[3], which is a set of constructs used to define meta-models. The MOF constructs include MOF class, MOF attributes, MOF association. These constructs literally constitute an ER representation (by using an Entity to represent a MOF class). Therefore, we believe ER representation is the right vehicle for representing intermediate meta-model. Moreover, meta-model using ER representation is easy to be described and serialized using XML [10]. This facilitates the meta-model exchange and processing using widely existent XML parsers.

[2] http://www.omg.org/mda/

[3] http://www.omg.org/cgi-bin/doc?formal/00-04-03

When modeling a WSDL for real business domain services implemented with a specific technology, we use the *generalization* relationship to extend those WSDL elements in Figure 2 rather than embedding the business domain service information as attributes to those WSDL elements. This avoids obfuscation of business and technology domain structure (actually meta-models of business/technology domain applications) with WSDL elements. The business domain information applies a generalization relationship to the operation entity, and technology domain information applies a generalization relationship to the binding entity. To exemplify, Figure 3 is a simple banking domain service specification.

Figure 2 shows the ER-based meta-model of this banking service WSDL. As can be seen from the figure, a typical business domain service represented as WSDL involves the extension of ER elements, which is associated to almost all the elements of WSDL. Nevertheless, by using the ER-based meta-model, such extension still keeps the original WSDL meta-model as shown in Figure 2 without being restructured, which helps generating WSDL from models with consistency.

2.2 The Mapping from ER based Meta-model to Other Forms of Meta-model

In GME, the containment relationship is represented by using a *model* element (tagged with <<*model*>>), which, in contrast to an *atom* element (tagged with <<*atom*>>), can contain other modeling elements. Also the contained elements can be promoted as *ports* of the model to have direct connections with external modeling elements. GME uses a *root model* as an entry point of access to all the modeling elements. Also, the *relationship* of ER is represented in GME as a first-class modeling element, *connection* (tagged with <<*connection*>>), with a *connector* in the form of a dot to associate this relationship with two modeling elements (entities).

The mapping from the ER-based meta-model to the counterpart in GME is based on the relationships in the ER representation. Three cases are involved as is shown in Figure 4. For the sake of limited space, below we only describe the mapping rules for case 3, i.e., B is specialized from A. In this case, A is rendered by an abstract FCO (First Class Object, tagged with <<*FCO*>>, represents an abstract generalization of other modeling constructs), a modeling element to be used as an abstract interface in GME, and B is represented as an inherited class of that FCO. Note there are two special treatments here: firstly, for the input/output elements of Figure 2, they are only used to tag the *connection* (named either "input" or "output") between message entities and its interconnecting entities in GME; secondly, the generalization

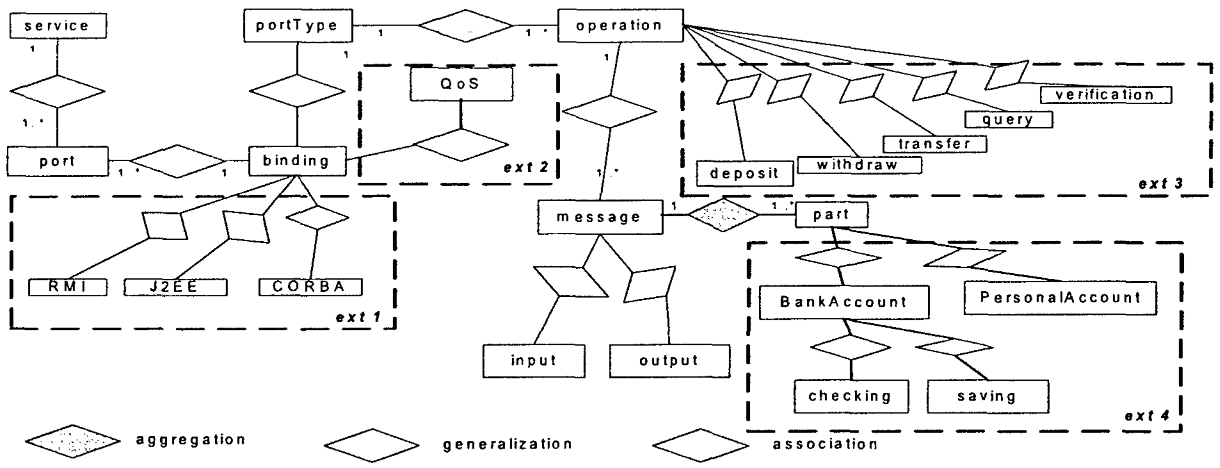


Figure 2: the ER-based Meta-model of Banking Service WSDL: the three parts enclosed with dashed line represent the extended part to the WSDL meta-model

```
A bank provides the service for users to set up accounts.  Account information includes personal data including Name, SSN, phone number, address,
and account data including Account Number, PIN, Transaction Record, Balance.  There are two types of accounts: checking account and savings
account.

For the bank side, it provides such services as: Account Verification, Account Query, Deposit, Withdraw, and Transfer.

The banking service implementation may use such technology as RMI, J2EE, and CORBA. Also it will enforce some Quality of Service (QoS)
requirements such as Availability, Dependability, Capacity.
```

Figure 3: the Banking Domain Service Description

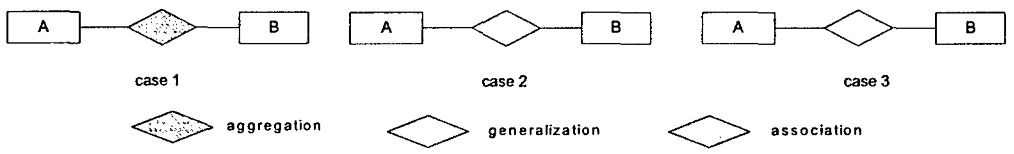


Figure 4: the Cases of Mapping from ER-based Meta-model to GME based Meta-model Based on the Relationship in ER Representation

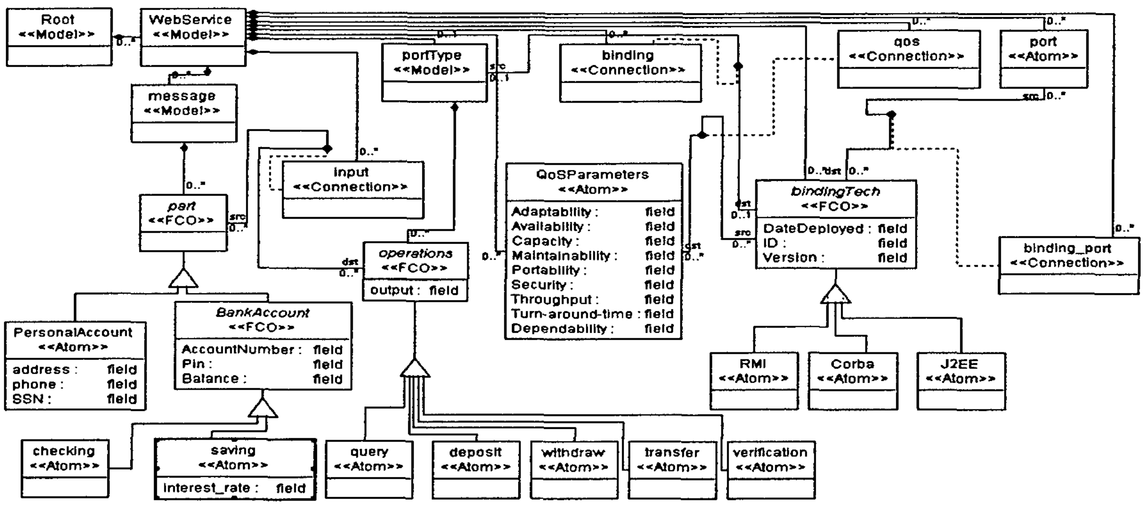


Figure 5: the Meta-model of Banking Domain WSDL in GME

relationship between binding and portType is actually treated as an association when modeling in GME, because the binding entity actually attaches values of the chosen protocol to the portType in WSDL rather than in the real sense of inheritance.

Figure 5 shows the meta-model created by mapping from the WSDL meta-model of the banking domain with ER representation to that in the GME strictly observing the above mapping rules. Note the model *WebService* corresponds to the service entity in Figure 2. The lower part of the models in Figure 5 are attributes for the related models to be instantiated in the modeling phase as described in the next section.

Based on this meta-model, a WS modeling environment can be constructed, and a generator based on this meta-model can be created to interpret WS models to generate WSDL. The WS modeling environment as well as generated WSDL is described in [1].

3. Related Work

In [6], MDA is used together with workflow technology for modeling and composing WS. But the authors do not provide a guideline as to how to create the meta-models. Also the mapping from PIM to PSM is not detailed. In contrast, we focus on meta-modeling WSDL only, while the meta-modeling approach is more complete and general. In [7], an MDA approach is used for BPEL code generation from a UML design. This approach uses XMI[4] processing technology for UML model exchange. Comparatively the XML representation for the ER model is much simplified and easy to process in our approach. Code generation in [7] is based on the UML profile mapping, which is not as flexible as a generator-based approach in our case.

4. Conclusion

WS domain-specific modeling environment provides a user-friendly environment to build WS with underlying WS-specific details abstracted. This paper presents a general meta-modeling approach to WS, which is used for the construction of WS domain-specific modeling environment In particular, we showed the merits of using the ER representation as an intermediate form for deriving and evolving meta-models to avoid the ad-hoc nature of constructing meta-models, which is an problem that is often not addressed (such as in [1]), particularly in constructing large-scale meta-models.

Meta-modeling of WSDL is a static structural modeling. Future work will include meta-modeling of WS behavior such as WS orchestration.

5. Acknowledgements

This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

6. References

[1] Cao, F., Bryant, B. R., Burt, C. C., Gray, J. G., Raje, R. R., Olson, A. M., Auguston, M., "Modeling Web Services: Toward System Integration in UniFrame," *Proc. 7th World Conference on Integrated Design and Process Technology (IDPT'03),* December, 2003, pp. 83-91.

[2] Chen, P. P., "The Entity-Relationship Model: toward a Unified View of Data," *ACM Trans. Database Systems*, March, 1976, pp. 9-36.

[3] Frankel, D. S., *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley, 2003.

[4] *GME 2000 User's Manual, Version 2.0,* ISIS, Vanderbilt University. 2001.

[5] Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J. and. Karsai, G., "Composing Domain-Specific Design Environments," *IEEE Computer*, November, 2001, pp. 44-51.

[6] Lopes, D., Hammoudi, S., "Web Service in the Context of MDA," *Proc. International Conference on Web Services (ICWS'03)*, June, 2003, pp 424-427.

[7] Mantell, K., "From UML to BPEL: Model Driven Architecture in a Web Services World," http://www-106.ibm.com/developerworks/ webservices/library/ws-uml2bpel/.

[8] Olson, A. M., Raje , R. R., Bryant, B. R., Burt, C. C., Auguston, M., "UniFrame-A Unified Framework For Developing Service-oriented, Component-based, Distributed Software Systems," to appear in *Service-Oriented Software System Engineering: Challenges and Practices*, ed. Zoran Stojanovic and Ajantha Dahanayake, 2004.

[9] Papazoglou, M. P., Georgakopoulos, D., "Service-Oriented Computing," *Commun. ACM,* October, 2003, pp. 25-28.

[10] Zhou, S., Xu., C., Wu, H., Zhang, J., Lin, Y., Wang, J., Gray, J. G., Bryant, B. R., "E-R Modeler: A Database Modeling Toolkit for Eclipse," *Proc. Annual ACM Southeast Conference*, April, 2004, pp.160-165.

[4] XML Metadata Interchange - http://www.omg.org/ technology/documents/formal/xmi.htm

Model-Driven Reengineering Legacy Software Systems to Web Services

ABSTRACT

The advancement of internet technology enables legacy software systems to be reused across geographical boundaries. Web Services (WS) have emerged as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA), the open standard description language XML and transportation protocol HTML. Therefore, legacy software systems can incorporate WS technology in order to be reused and integrated in a distributed environment across heterogeneous platforms. In this paper, we present a comprehensive, systematic, automatable approach toward reengineering legacy software systems to WS applications, rather than rewriting the whole legacy software system from scratch in an ad-hoc manner.

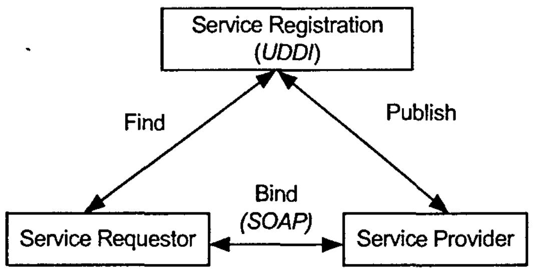
Keywords: software system reengineering; Web Services; Model-Integrated Computing; meta-model; model; model marshaling and unmarshaling; Entity-Relationship model

INTRODUCTION

Web Services as a Presentation Layer for Legacy Software Reuse and Integration

With the rapid advancement of software technology, more and more software systems developed with the state-of-the-art technologies of yesterday are becoming legacy software systems of today. Specifically, we define legacy software in a comparative manner, i.e., the software systems are *legacy* if the languages, models or platforms they are developed with can be replaced with new languages, models or platforms of advanced features and improved capabilities. The reuse and integration of legacy software systems offer a promising direction for boosting productivity by dramatically reducing both cost and time-to-market expenses (Devanbu et al., 1996). With the emergence and advancement of Internet technology, the power of legacy software systems is being unleashed toward a broader scope. Particularly, Web Services (WS) have emerged as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA) (Colan, 2004) as is illustrated in Figure 1. By using standard XML as the description language and HTTP as the transport protocol, WS can be used to wrap legacy software systems for integration beyond the enterprise boundary across heterogeneous platforms. To be specific, WS uses the XML based XML-based Web Services Description Language (WSDL) for specifying services, SOAP (Simple Object Access Protocol) messages for service invocation, and UDDI (Universal Description, Discovery and Integration) registry for service discovery (Colan, 2004). With the wrapping by WS, the integration of legacy software systems is simplified, from one to one interoperation to interoperate on the one common ground (WS).

Figure 1. Service Oriented Architecture (SOA)



Approaches for Using Web Services as a Wrapper

There are several options for reengineering legacy software to WS:

- Manually port original software source code to WS applications. This is an expensive solution. Also WS code, such as WSDL, is verbose, and coding WSDL manually is error prone.

- Language tool based—in which the legacy software package is recompiled to generate WSDL. Many tools such as AXIS[i], and the Microsoft .Net framework provide the function of generating WSDL from implementation code (such as Java and C#) and vice versa. Such tools leverage compiler technology to generate WSDL from other programming languages. The WSDL in turn can be used to generate client side stub code for the client to call the services exposed by legacy software systems (Graham, 2002). However, this language tool based solution remains to be language-dependent. With the variety of legacy software systems, a language neutral solution is required in order to sufficiently handle the reengineering of legacy software systems to WS.

Cao, et al. (2004) used a model-driven approach to WS development. We build upon this work by presenting a model-driven approach for reengineering legacy software systems to the WS applications, in which a model plays a central role for migrating legacy software systems to WS implementations. A model is usually represented in UML[ii], or any other abundant domain specific visual language (as can be seen in JVLC[iii]), which represents the structural and contextual information of a legacy software system in a language neutral style without being tied to implementation specifics. The model-driven reengineering approach is also based on the observation that legacy software systems are usually documented in a visual modeling language; models can also be used as first-class assets in SOA (e.g., model as the basis for service discovery in Hausmann, et al., 2004).

To apply the model-driven approach for reengineering legacy software systems to WS, a model should play a role beyond the conventional design and documentation capacity, i.e., a role for WS code generation directly to resolve the manual porting problem as described above. Usually UML-based code generation is based on a static mapping from the UML profile (Frankel, 2003), which lacks flexibility during code generation process. As such, we use Model IntegratedComputing (MIC) (Lédeczi et al., 2001) for building a WS modeling environment and consequently for WS code generation. MIC is essentially a development paradigm that offers a means for creating a modeling language (*meta-model*), its associated modeling language interpreter (*generator*). Then any domain-specific model built based on the modeling language can be interpreted by traversing the model tree. The result of the interpretation process is the code synthesized from the model. MIC has been widely used in middleware (Gokhale et al., 2004; Edwards et al., 2004) and embedded systems (Karsai et al., 2003; Lédeczi et al., 2003).

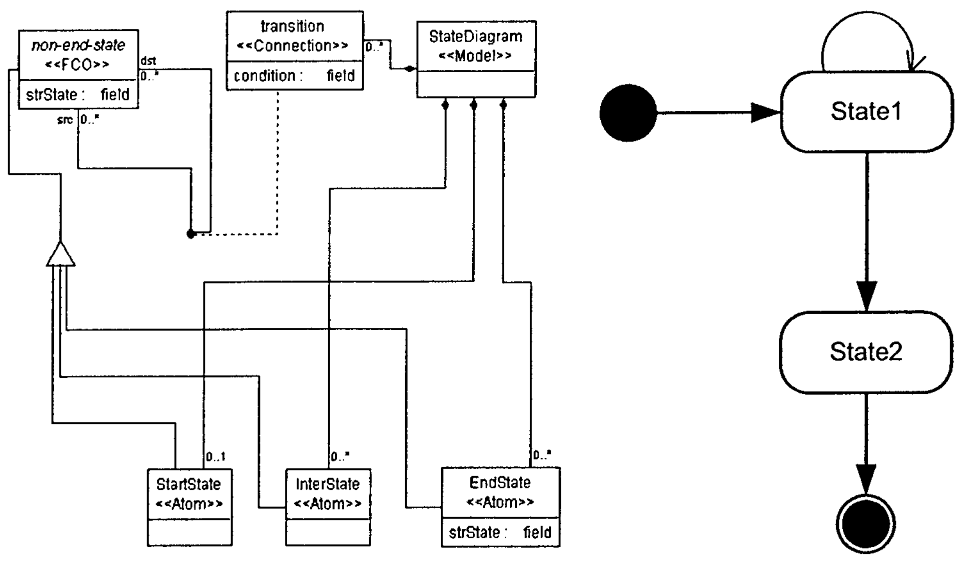
Table 1. Comparison between MIC and programming language

MIC	**Programming Language**
meta-model	grammar
generator	compiler/interpreter
domain-specific model	application developed using the corresponding language
code synthesized in any chosen language	intermediate code or native code

To ease the understanding of MIC, Table 1 provides an analog between MIC and conventional programming language elements. Figure 2 provides an example of a meta-model of Finite State Machine (FSM) and the corresponding model based on it.

While the meta-model (and in the later part the domain-specific modeling environment) described in this paper is based on the notation of the Generic Modeling Environment (GME) (ISIS, 2001) (as it is the only tool for the MIC paradigm so far), the same principle as shown in this paper can be applied to other MIC-compliant modeling tools as well.

Figure 2. A simple example of meta-model and model



Finite State Machine (FSM) Meta-model *Finite State Machine Model*

Problems for Applying Model Integrated Computing (MIC) to Reengineering Legacy Software to WS

While MIC offers an automatable and language neutral approach for reengineering legacy software to WS, the starting point of MIC - the construction of the meta-model has to be a manual process. Previous work on WS modeling (Cao et al., 2003) has revealed that with the increasing complexity of the modeling target, the construction of the meta-model is subject to being ad-hoc and error-prone. With the modeling assets (UML or other domain specific visual modeling language) already abundantly available as part of the legacy software (which we term *legacy model*), it is desirable to derive the meta-model from the legacy model in a systematic, automatable process as opposed to being ad-hoc and error-prone. However, the current meta-modeling languages lack adequate modularity support for large scale meta-model construction, which nevertheless is widely existing in general programming languages. As a result, the construction of a meta-model remains an art rather than a science.

Therefore, this paper is composed of two major parts, each corresponding to the primary contributions of this paper:

1) the elicitation of a meta-model from a legacy model in a systematic, automatable process, which is addressed in Section 2 and Section 3, and consequently
2) the creation of a domain-specific WS modeling environment for WS code generation in Section 4, as well as the treatment of WS semantic concerns from a model-driven perspective in Section 5.

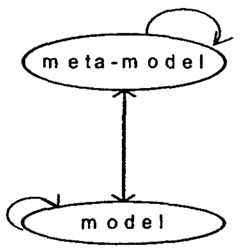
Related work is described in Section 6, followed by the conclusion and future work in Section 7.

MARSHALING AND UNMARSHALING MODELS USING THE ENTITY-RELATIONSHIP (ER) MODEL

The elicitation of a meta-model from UML or other domain-specific modeling notations can be done on a per source model basis. However, with the constant emergence of new modeling notations, the elicitation approaches will become ad-hoc and not reusable. Moreover, there is a need to converge the diversified modeling assets for modeling tool integration[iv]. Therefore, we need to encode the diversified models with a common representation, such that different modeling notations can transfer to and from it, thus modeling assets can be exchanged and used across different modeling tools. Cao et al. (2005) have referred to these modeling notation transferals as *marshaling* and *unmarshaling*, respectively. The term marshaling comes from the distributed computing scenario where heterogeneous data types are always translated into some common data type over the network so as to be consumed at another end of the distributed environment, where the common data type is unmarshaled again into another environment-specific data type. Comparatively, the concept of marshaling and unmarshaling models refers to transform a model to an *intermediate common semantic* form, which is reinterpreted in another modeling environment/tool. This intermediate common semantic form is in a similar vein to

ACME (Garlan et al., 2000), which is an intermediate form for exchanging software architecture description languages across different software architecture design tools. Moreover, with the heterogeneity of models at different meta-level (not only model level but also meta-model level) (Frankel, 2003), marshaling and unmarshaling of models can be performed at different levels: horizontally, meta-model level and model-level; vertically, meta-model to/from model as is illustrated in Figure 3.

Figure 3. Marshaling and unmarshaling models at different levels: the arrow represents marshaling/unmarshaling process



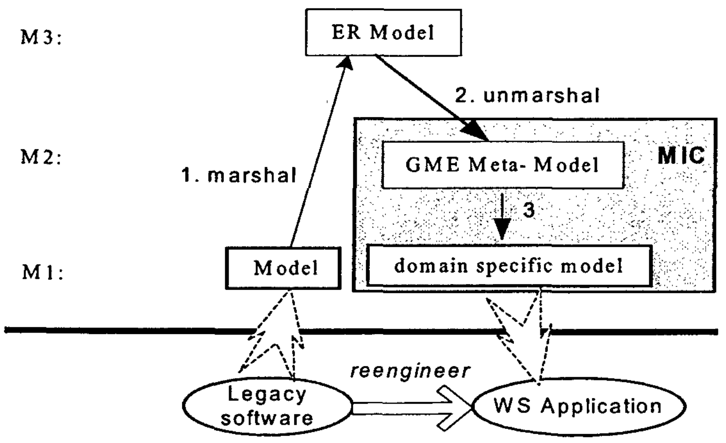
Here we use the ER model (Chen, 1976) as the intermediate common semantic form for marshaling and unmarshaling models[v]. The rationales are as follows:

- **Sufficiency.** Even though UML is widely adopted in software modeling, which seems to justify the use of UML as a common model for exchanging model assets across modeling facilities, UML is not convenient for model serialization, thus not fit for modeling asset exchange, reuse and evolution. In fact, the object diagram (Booch et al., 1999), for which UML is used to capture and store the snapshot of software system state, is represented virtually in an Entity (object) and Relationship (links) model. Moreover, the UML modeling language has its roots in the ER model, and the latter is already widely used as the foundation for CASE tools in software engineering and repository systems in databases[vi].

- **Necessity.** As is illustrated in Figure 3, not only models, but also meta-models are in need of marshaling and unmarshaling. Therefore, the intermediate model should be expressive enough to be at the meta-meta model level in the meta-level stack (Frankel, 2003). The meta-meta-model is described by the Meta Object Facility (MOF)[vii], which is a set of constructs used to define meta-models. The MOF constructs are the *MOF class*, the *MOF attributes* and the *MOF association.* These constructs correspond to an ER representation (by using an Entity to represent a MOF class), which indicates that the ER representation is semantically equivalent to MOF fundamentally. Therefore, the ER representation is the right vehicle to play the dual roles of marshaling both models and meta-models. Also, other non-UML based languages, even though not as popular, are abundantly present, for which UML is not an omnipotent cure.

The scope of this paper is on vertical direction which is further illustrated in Figure 4, i.e., marshaling models to ER model, then unmarshaling ER model to the GME meta-model. The gray area in Figure 4 represents the MIC paradigm. To be specific, in the following section, we will marshal a UML class diagram for Web Services Description Language (WSDL) to the GME meta-model, then create a WS modeling environment based on the meta-model for WS code generation. Therefore, legacy software systems can be reengineered to the WS application automatically with a language neutral approach. We also show the generality of this approach: even though the scope is within the vertical direction, the approach can also be applied for horizontal marshaling/unmarshaling using ER model; even though the source model is the UML object-oriented model, it is not tied to this single kind of source model and can be applied to other domain-specific visual modeling languages as well.

Figure 4. Eliciting Meta-models from model via marshaling and unmarshaling models using ER model



REENGINEERING LEGACY SOFTWARE TO WEB SERVICES (WS)

In order to reengineer legacy software to WS, we need to capture 1) the WS technology domain knowledge; 2) the original legacy software business domain knowledge; and 3) original implementation technology information. This categorization of technology domain knowledge and business domain knowledge has been described by Zhao, et al. (2003).

Figure 5 is the class diagram of WSDL. The WS ***message***s, which are either ***input*** or ***output*** messages, are composed of ***part***s, each of which corresponds to a specific data ***type***. The ***portType*** is an abstract WS interface definition, where each contained element, i.e., the ***operation***, defines an abstract method signature. The operation uses messages as its parameters. ***Binding*** represents an instantiation to the abstract ***portType*** with concrete protocol and data type. ***Service*** is a collection of ***port***s, denoting a deployment of a binding at a specific network location.

Figure 5. The architecture of WS description elements

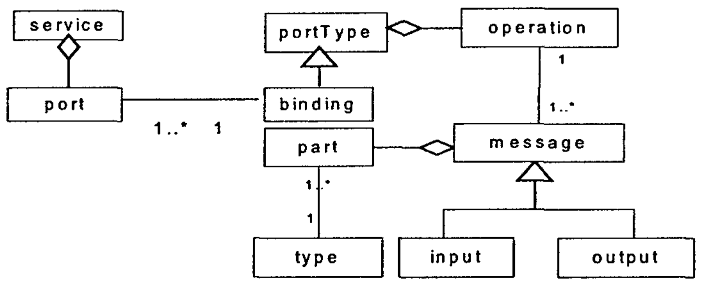


Figure 6 describes the legacy banking application information, including its business domain knowledge (the first two paragraphs) and its original technology domain knowledge (the last paragraph). Note as WS is used as wrapper for original technology domain knowledge together with the business domain knowledge, rather than replacing the original technology, we treat the original domain knowledge as the part of business domain knowledge in the remaining part of the paper for simplicity purpose.

Figure 6. A banking example

A bank provides the service for users to set up accounts. Account information includes personal data including Name, SSN, phone number, address, and account data including Account Number, PIN, Transaction Record, Balance. There are two types of accounts: checking account and savings account.

For the bank side, it provides such services as: Account Verification, Account Query, Deposit, Withdraw, and Transfer.

The banking service implementation may use such technology as RMI[viii], J2EE[ix], and CORBA[x]. Also it will enforce some Quality of Service (QoS) requirements such as Availability, Dependability, Capacity.

Marshaling Legacy Software Model to ER Model

In order to elicit the banking domain WS meta-model, we need to first merge the WS technology domain information with the business domain information. To that end, we treat the WS technology domain as the *dominant domain* during the merge process, with the business domain knowledge as the *adjunct domain* being appended to the marshaled model from the technology domain model. As such, the marshaling process as illustrated in Figure 4 can be decomposed into the marshaling type A for dominant domain and type B for adjunct domain together with a merge step as is illustrated in Figure 7.

Figure 7. Stepwise marshaling

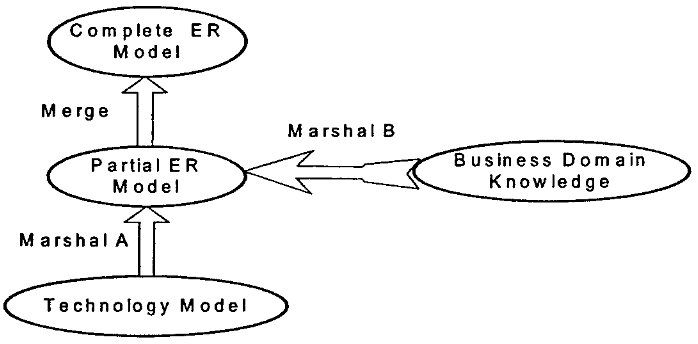
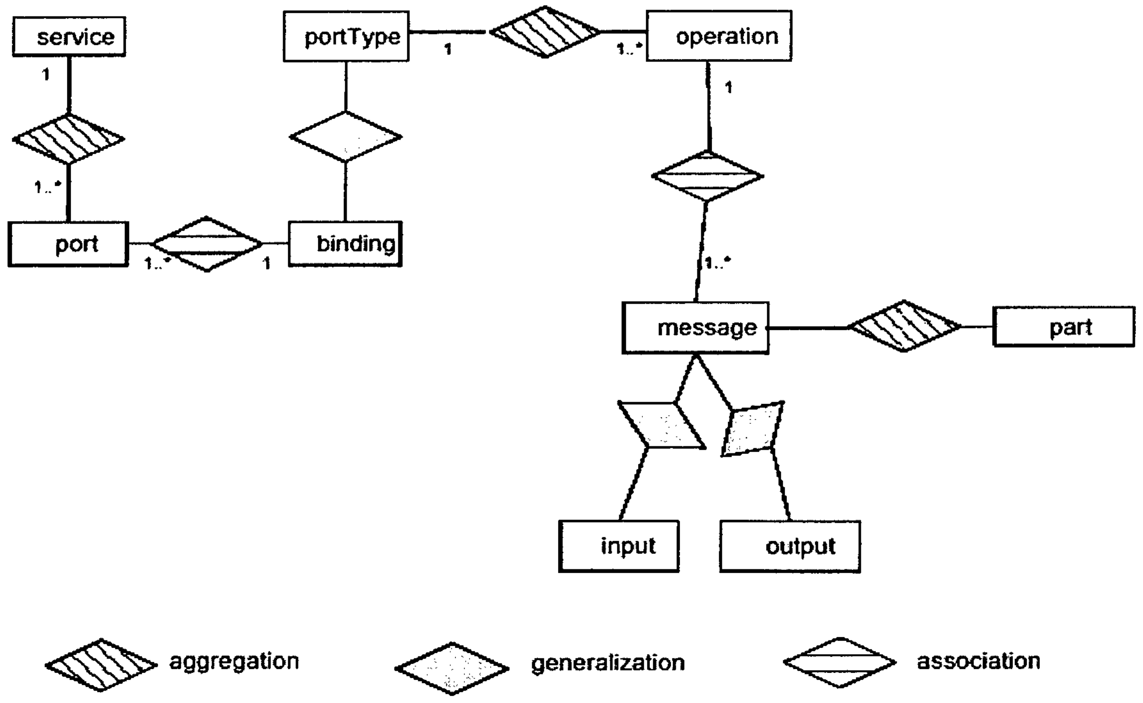


Table 2. Marshaling rules

Type	Rule
Marshal A	▪ aggregation, association, generalization, and dependency => *Relationship* ▪ class=> *Entity*
Marshal B	domain analysis and mapping

Table 2 illustrates the marshaling rules based on different marshaling types. Note that one of the essential characteristics of a meta-model is that it treats not only the models, but also the inter-relationships among models as first-class entities. Therefore, for marshal type A, the different type of relationships between classes will be mapped to the *Relationship* construct in the ER model, while each class is represented as an *Entity*. Figure 8 illustrates the resultant ER model after marshaling the WS class diagram based on this rule. Each diamond represents a type of relationship in the original class diagram. Note we ignore *type* in the ER model of Figure 5,

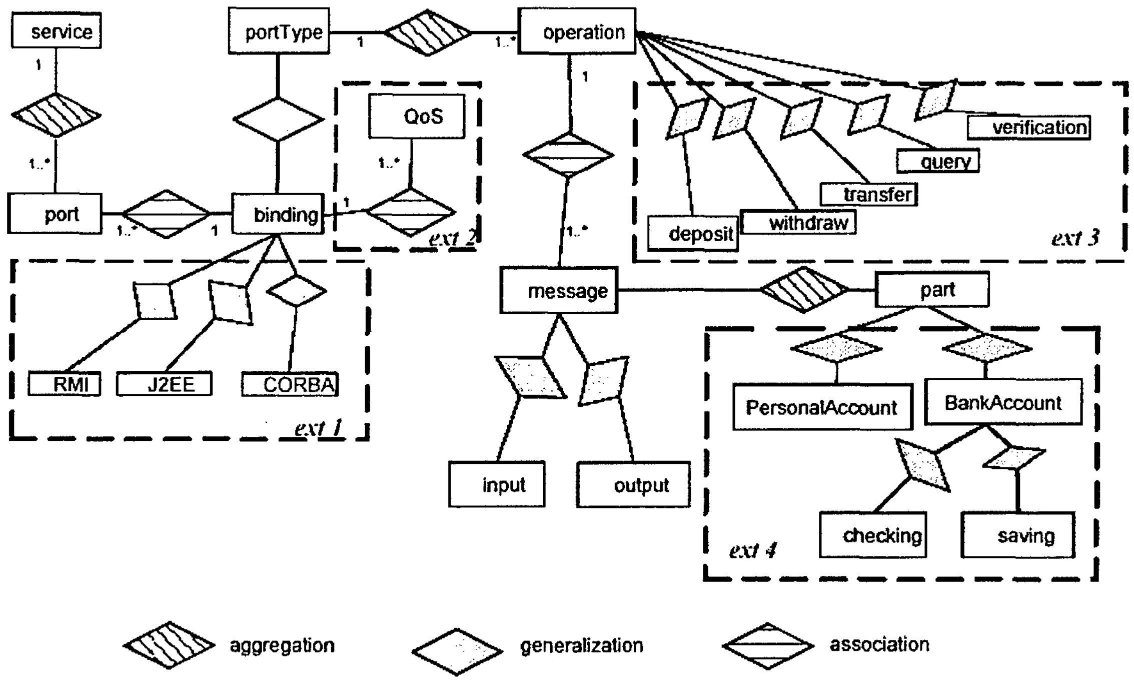
Figure 8. Marshaling WSDL model to ER model



because we can put the type directly as the attribute of the part element. However we will not include the attributes to the entities and relationships in the ER representation here, as the focus of this paper is about the model of marshaling and unmarshaling structurally; the attributes will be annotated in the GME meta-model and are shown later.

For marshal type B, a domain analysis phase (Czarnecki & Eisenecker, 2000) is needed to associate the business domain information to the technology domain information. Specifically, the different banking services described in Figure 6 can be treated as different types of **operations** in WSDL, while different banking service implementation technology and QoS requirements can be associated to **bindings** in WSDL as a reification of operations. Account information and account type information can be treated as **messages** in WSDL. Figure 9 illustrates in detail the resultant ER model after annotating the business domain knowledge (using either generation relationship or association relationship) to the WSDL ER model illustrated in Figure 8. By using the ER model as the intermediate form for marshaling, different types of

Figure 9. The ER model of Banking Service WSDL: the three parts enclosed with dashed line represent the extended part to the WSDL model.



domain knowledge can be merged incrementally without obfuscating each other, which provides a separation of concerns toward domain-specific model refinement. Also with the non-invasive merge process, the business domain semantics are reified with technology semantics while the business domain semantics are kept unchanged.

Just as the compiler can apply code optimization when compiling application code, the marshaling process can be used to apply optimization (e.g., reduce redundant models or relationships) for the original modeling language (either UML or domain specific), the detailed discussion of which is out of the scope of this paper.

Unmarshaling ER Model to GME Meta-model

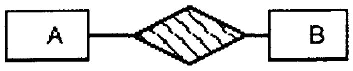
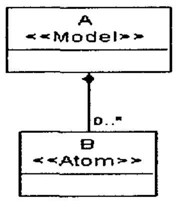
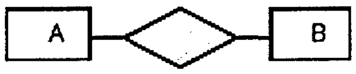
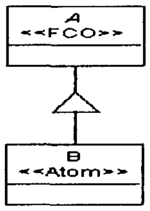
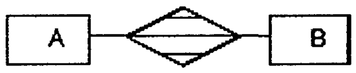
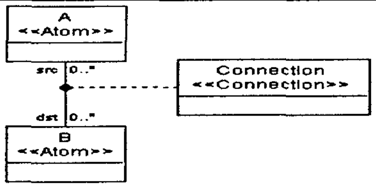
In the GME meta-model, the containment relationship is represented by using a *model* element (stereotyped with <<*model*>>), which, in contrast to an *atom* element (stereotyped with <<*atom*>>), can contain other modeling elements. Also the contained elements can be promoted as *ports* of the model to have direct connections with external modeling elements. Additionally, GME uses a *root model* as an entry point of access to all the modeling elements. Also, the *relationship* of ER is represented in GME as a first-class modeling element, *connection* (stereotyped with <<*connection*>>), with a *connector* in the form of a dot to associate this relationship with two modeling elements (entities).

The unmarshaling from the ER model to the GME meta-model is based on the relationships in the ER representation, as is illustrated in Table 3.

1) A contains B. In this case, A can be modeled as a *model* element in GME containing B.

2) B is specialized from A. In this case, A is rendered by an abstract FCO (First Class Object, tagged with <<*FCO*>>, represents an abstract generalization of other modeling constructs), a modeling element to be used as an abstract interface in GME, and B is represented as an inherited class of that FCO. Note there are two special treatments here: first, for the input/output elements

Table 3. The Unmarshaling Rules: the relation notation is consistent with that in Figure 8

Rule Number	Relationship type	GME Metamodel element
1		
2		
3		 A <<Atom>> src 0..* Connection <<Connection>> dst 0..* B <<Atom>>

of Figure 9, they are only used to tag the *connection* (named either "input" or "output") between message entities and its interconnecting entities in GME; second, the generalization relationship between binding and portType is actually treated as an association when modeling in GME, because the binding entity actually attaches values of the chosen protocol to the portType in WSDL rather than in the real sense of inheritance.

3) B is associated to A. In this case, a *connection* can be added to be associated with the A and B representations in GME. The connection element can be named with respect to A's or B's properties as a kind of tag, e.g., the tag can be named as the combination of both A's name and B's name. Note when the situation as described in case 2 applies, then this tag should be named as in case 2.

Figure 10. The meta-model of banking domain WSDL in GME

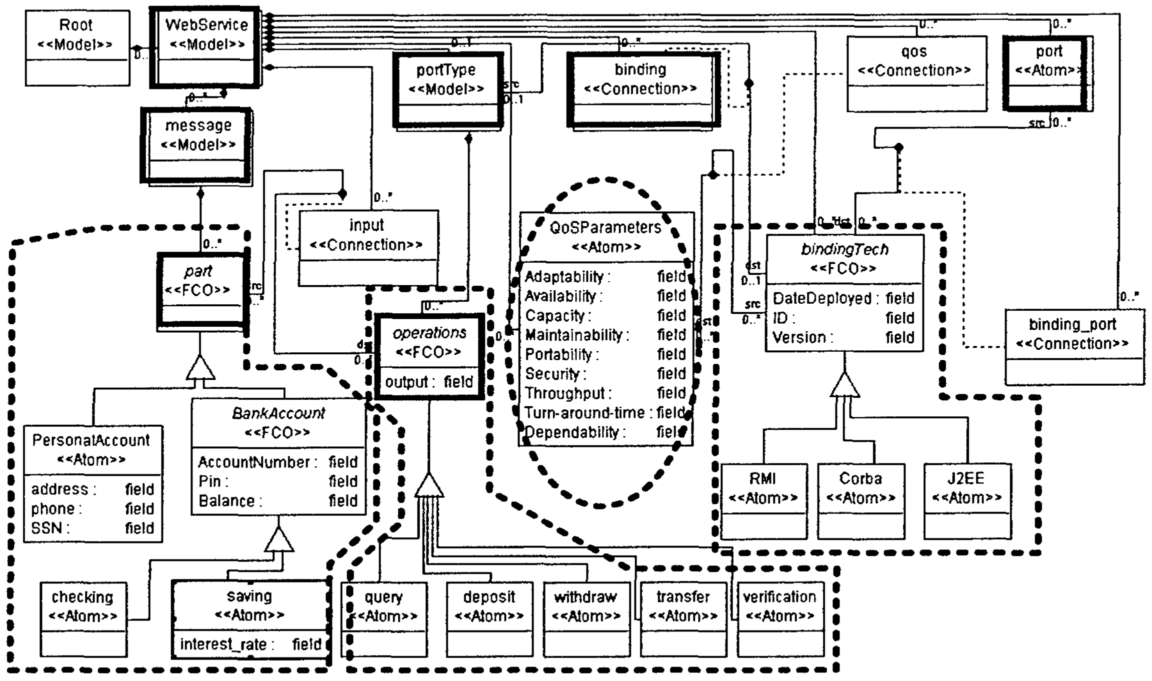
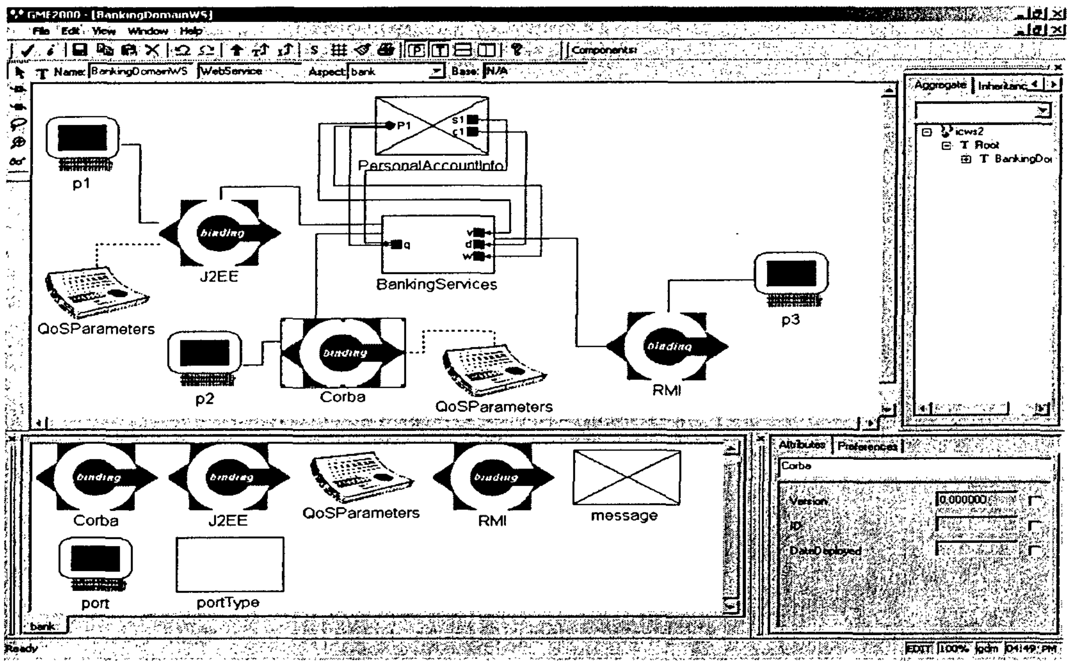


Figure 10 shows the meta-model created by unmarshaling the ER model in Figure 9 strictly observing the above unmarshaling rules. The seven boxes with bold borders correspond to the seven WSDL entities in Figure 8 and 9, with *WebService* corresponds to the *service* entity. The boxes in Figure 10 also contain attributes for the related models to be instantiated in the modeling phase. The four areas designated by four bold dashed circular lines correspond (from right to left) to the extension parts 1-4 in Figure 10. It can be seen from Figure 10 that the meta-modeling language lacks the modularity that programming languages have, thus the construction process of a complex meta-model is error-prone without a systematic, automatable treatment.

THE WS MODELING ENVIRONMENT

After a meta-model is derived by marshaling and unmarshaling models, a domain specific modeling environment (which is also a crucial part of MIC) can be created based upon the meta-model, as is indicated in Table 1. Figure 11 shows the screenshot of the banking-domain WS modeling environment based on the meta-model illustrated in Figure 10. The lower-left corner provides the modeling elements that can be dragged and dropped in the upper-left pane for

Figure 11. The banking domain-specific WS modeling environment



constructing a banking service model. The names of the models in the lower-left pane represent the meta-model names (*kind names*); when those models are dragged to the above pane, the model name can be changed to reflect the meaning of the model in the domain-specific context, which we call a *context name*. Furthermore, the domain-specific model can be traversed based on the meta-model and interpreted in terms of code generation using the GME Builder Object Network (BON) framework (ISIS, 2001), which is illustrated in Figure 12. For saving space, Figure 12 only shows the interpreter code for generating message and portType of WSDL. Other part of WSDL can be generated in a similar way. The WSDL code generated for the banking service embedded with QoS parameter extension is shown in Figure 13. Because of the limited space, only a snippet of the generated WSDL code is shown in Figure 13. Notice the bold-font part of the following WSDL code includes the QoS and ontology attributes of WSDL, which may be used for WS filtering if QoS requirements or domain specific requirements are include for service discovery.

Figure 12. WSDL code synthesis using GME BON API

```
const CBuilderModelList *root = builder.GetRootFolder()->GetRootModels();
POSITION pos = root->GetHeadPosition();
ASSERT(pos->GetCount()==1);  //to ensure this model is representing just one WSDL

CBuilderModel *webserv = pos->GetHead(); //get the handle to the WebService model
ASSERT(webserv->GetKindName()=="WebService");

//WSDL message part
const CBuilderAtomList *messages = webserv->GetModels("message");
pos=messages->GetHeadPosition();
CBuilderAtom *oneMessage;
while(pos)
  {
    /*
     traverse each message model and generating code
     <message>... </message>
     for each message model
    */

     oneMessage=messages->GetNext(pos);
     const CBuilderAtomList *accounts =oneMessage->GetAtoms("PersonalAccount");
     ...
  }

//WSDL portType part
const CBuilderAtomList *portType = webserv->GetModels("portType");
pos=portType->GetHeadPosition();
ASSERT(pos->GetCount()==1);  //to ensure only one portType element in WSDL
CBuilderAtom *oneportType;
oneportType=portType->GetNext(pos);
.....
}
```

Figure 13. The WSDL for a banking WS

```
<message name="checking">
 <part name="user_ident" type="identity"/>
 <part name="p1" type="checking"/>
</message>
<message name="savings">
 <part name="user_ident" type="identity"/>
 <part name="p1" type="savings"/>
</message>
<message name="checking_savings">
 <part name="user_ident" type="identity"/>
 <part name="p1" type="checking"/>
 <part name="p2" type="savings"/>
</message>

<portType name="BankingServices">
    <operation name="w"
             ontology="Banking:withdraw">
      <input message="checking"/>
      <output message=""/>
    </operation>
   <operation name="d"
             ontology="Banking:deposit">
       <input message="checking"/>
       <output message=""/>
   </operation>
  <operation name="v"
             ontology="Banking:deposit">
   <input message="checking_savings"/>
   <output message=""/>
  </operation>
  <operation name="q" ontology="Banking:query">
     <input message="savings"/>
      <output message=""/>
  </operation>
</portType>
```

(to be continued in the right pane)

```
<binding name="J2EE_Banking"
            type="BankingServices">
  <soap:binding style="J2EE" transport="http"
    QoS:portability="0.544400">
    .........
</binding>
<binding name="CORBA_Banking"
            type="BankingServices">
  <soap:binding style="CORBA" transport="IIOP"
    QoS:turn-around-time="10.35">
    .........
</binding>
<binding name="RMI_Banking"
            type="BankingServices">
  <soap:binding style="RMI" transport="http"
    QoS:dependability="0.34">
    .........
</binding>

<service name="My Bank">
  <port name="p1" binding="J2EE_Banking">
      <soap:address location="URL1"/>
  </port>
  <port name="p2" binding="CORBA_Banking">
      <soap:address location="URL2"/>
  </port>
  <port name="p3" binding="RMI_Banking">
      <soap:address location="URL3"/>
  </port>
</service>
```

MODEL-DRIVEN APPROACH TO ENRICH WS SEMANTICS

Current WS standards mainly embrace the semantics of processes at the collaborating syntactic interface level. WSDL only exposes distributed object services, while such process behavior aspects as ordering, and dependency are not well specified in the existing WSDL standard. The model-driven approach can play a unique role in enriching the WS semantics:

- OCL (Object Constraint Language)[xi] to enrich WS semantics at a high level

  OCL is used to complement the semantic representation for UML. Likewise, when the model is used to represent WS, OCL can be used to enrich WS semantics indirectly at a higher level. For example, if we add into the banking case in Figure 6 such requirement that "deposit and withdraw can only be applied to checking account", the specified constraints over withdraw and deposit operations can be enforced in GME using the following MCL expression (ISIS, 2001), an OCL implementation in GME:

  ```
  connectedFCOs("src")->forAll(c|c. kindName()="checking")
  ```

  Those constraints apply to both the withdraw atom and the deposit atom in Figure 10, which means those First Class Objects (referring to both entities and relations in GME) that are connected with withdraw/deposit atoms are all of kind "checking". Therefore, in the WS modeling environment as shown in Figure 11, once a modeling entity of type other than "checking" is connected to withdraw/deposit, an error message window will pop up.

- Meta-model as Ontology

  A valid meta-model is an ontology, but not all ontologies are modeled explicitly as meta-models (Ernst, 2002). This ideal has already been used in (Hausmann et al., 2004) for WS discovery. Comparatively, here we just output the meta-model information into the generated WSDL as ontology annotation to enrich the WSDL semantic representation.

- Creating modeling language for enriching WS semantics

Assume there is order restriction for those banking operations described in Figure 6: both *transfer* and *withdraw* have to be preceded by a *query* operation; the *account verification* comes after each of the other operations. Such models as Finite State Machine (FSM) can be used to enrich WS semantics. Based on the FSM meta-model in Figure 2, a FSM modeling environment can be created in addition to the WS modeling environment that is described in Section 4, which can be used to generate operation ordering constraint code to be embedded in WSDL. We skip the details here due to space limitations.

RELATED WORK

This paper presents both a novel model-driven approach in general and its novel application to WS in particular. Specifically:

1) For the model-driven approach aspect, we use ER model for marshaling and unmarshaling models. The related work in this regard includes:

- **MDA**

  MDA[xii] is an initiative from OMG[xiii] for capturing the essence of a software system in a manner that is independent of the underlying implementation platform. MDA can assist in reengineering legacy software systems into Platform Independent Models (PIMs). A PIM can be mapped to software components on Platform Specific Models (PSMs), such as CORBA, J2EE or .NET. In this way, legacy systems can be reintegrated into new platforms efficiently and cost-effectively (Frankel, 2003). However, the core part of mapping technology for MDA is either ad-hoc or pre-mature before MDA can be fully adopted in industry. ER-based model marshaling and unmarshaling offers a potential solution to address this problem systematically. Another difference is that in MDA, the PIM is treated as dominant model while here we treat the technology domain as dominant model, with business domain knowledge (PIM) as adjunct model in Section 3.

It has been observed that the ER representation has been adopted in defining the Knowledge Discovery Meta-Model (KDM)[xiv] and Ontology Definition Meta-Model (ODM)[xv] in OMG, which underscores the role that ER plays for model marshaling and unmarshaling.

- **Grammar Inference**

  The ER model, because of its powerful modeling capacity, can be used as an intermediate form for model-to-model and meta-model-to-meta-model exchange. Because of the dual role that the ER model can play, it is treated as an intermediate form for model-to-meta-model elicitation, which is the theme of this paper. This idea is very similar to grammar inference (Higuera, 2001), where a grammar can be inferred from language examples. But the two approaches are applied at different abstraction levels.

- **XMI**

  XMI[xvi] provides a standard mapping from MOF-based models to XML, which can be exchanged between software applications and tools, and the XMI specification is difficult to read by humans. In contrast, ER-based model marshaling and unmarshaling represents a design-level approach for evolving design assets, without being restricted to low-level syntactical data representation specifics, and the ER representation is much more human comprehensible. Also, the XMI-based approach uses top-down mapping, and is coupled to the meta-model of the targeted language; interchange format cannot be changed without changing the meta-model. In contrast, the ER-based approach represents either horizontal mapping or bottom-up mapping as is illustrated in Figure 3, without being tied to any meta-model.

2) We applied the model-driven approach to WS, specifically, MIC for WS code generation automatically; Model-driven approaches for enriching WS semantics are also identified. The related work in this regard is as follows:

In Lopes and Hammoudi (2003), MDA is used together with workflow technology for modeling and composing WS. But the authors do not provide a guideline as to how to create the meta-models. Also the mapping from PIM to PSM is not detailed. In contrast, our meta-modeling approach is sufficiently complete and general as to be applicable to other aspects of WS such as WS orchestration code generation. Sivashanmugam (2003) describes an approach of adding semantics to WS by adding ontology attributes to both WSDL and UDDI, which includes pre-condition and effect specification. We applied ontology annotation to WS as well, and we put the pre-condition and other effect specification at the meta-model level. In Mantell (2003), an MDA approach is used for BPEL4WS[xvii] code generation from a UML design. This approach uses XMI processing technology for UML model exchange. Comparatively, the XML representation for the ER model is much simpler and easier to process in our approach. Code generation in Mantell (2003) is based on the UML profile mapping, which is not as flexible as a generator-based approach in our case.

The UniFrame project (Raje et al., 2002; Olson et al., 2004), has a more comprehensive application of the model-driven approach. UniFrame aims at creating a framework for seamless integration of distributed heterogeneous components. In UniFrame, the model-driven approach is applied for domain engineering, and for creation of Generative Domain Models (GDMs) (Czarnecki and Eisenecker, 2000), which are used for eliciting rules to generate glue/wrapper code for assembling distributed heterogeneous components. In contrast, the scope of glue/wrapper code generated here is specific to WS code, which has not been addressed by UniFrame.

CONCLUSION AND FUTURE WORK

With Web Services (WS) as a wrapper, legacy software systems can be reused and integrated beyond enterprise boundaries across heterogeneous platforms. This paper explores in detail a model-driven approach to reengineer legacy software system to WS applications using a systematic, automatable process, which includes: 1) the meta-modeling process using ER-based marshaling and unmarshaling, 2) the construction of a WS modeling environment for generating

WS code and enriching WS semantics. To our best knowledge, there is no peer work that addresses either systematic meta-model construction, or sufficient model-based WS code generation, while our work represents a comprehensive solution to both issues. Even though the work presented in this paper is specific to WS development, the approach can be applied to other web system engineering by reengineering to a different meta-model other than the WS meta-model.

Future work will be to provide tool support for part 1 in the preceding paragraph to automate the model marshaling and unmarshaling process for seamlessly integrating the reengineering process to MIC paradigm. For part 2, we will enrich the WS modeling environment by providing modeling and code generation support to other behavior concerns of WS such as interaction, activity, and temporal relationship, as well as WS orchestration and adaptation.

ACKNOWLEDGEMENTS

This research is supported in part by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

REFERENCES

Booch, G., Rumbaugh, J. & Jacobson, I.(1999). The Unified Modeling Language User Guide. Addison-Wesley.

Cao, F., Bryant, B. R., Burt, C., Gray, J., Raje, R., Olson, A., & Auguston, M. (2003). Modeling Web Services: toward system integration in UniFrame. *Proceedings of 7th World Conference on Integrated Design and Process Technology (IDPT'03).*

Cao, F., Bryant , B. R., Zhao, W., Burt, C., Gray, J., Raje, R., Olson, A., & Auguston, M. (2004). A Meta-modeling approach to Web Services. *Proceedings of 2004 IEEE International Conference on Web Services (ICWS 2004).*

Cao, F., Bryant , B. R., Zhao, W., Burt, C., Gray, J., Raje, R., Olson, A., & Auguston, M. (2005). Marshaling and unmarshaling models using Entity-Relationship model. *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC 2005).*

Chen, P. P. (1976). The Entity-Relationship model: toward a unified view of data. ACM Transactions on Database Systems, 1(1), 9-36.

Colan, M. (2004) Service-oriented architecture expands the vision of Web Services. http://www-106.ibm.com/developerworks/webservices/library/ws-soaintro.html.

Czarnecki, K., & Eisenecker, U.W. (2000). Generative Programming: Methods, Tools, and Applications. Addison Wesley.

Devanbu, P., Karstu, S., Melo, W., & Thomas, W. (1996). Analytical and empirical evaluation of software reuse metrics. *Proceedings of 18th International Conference on Software Engineering (ICSE'96).*

Edwards, G. T., Deng, G., Schmidt, D. C., Gokhale, A. S., & Natarajan, B. (2004). Model-driven configuration and deployment of component middleware publish/subscribe services. *Proceedings of 3rd international Conference on Generative Programming and Component Engineering (GPCE 2004).*

Ernst, J. (2002). What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model? http://www.metamodel.com/article.php?story=20030115211223271.

Frankel , D. S. (2003). Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley.

Garlan, D., Monroe,R. T., & Wile, D. (2000). Acme: architectural description of component-based Systems. Foundations of Component-Based Systems, ed. Leavens, G. T. and Sitaraman, M., Cambridge University Press, 47-68.

Gokhale, A., Schmidt, D. C., Natarajan, B., Gray, J., & Wang, N. (2004) Model driven middleware. Middleware for Communications, ed. Mahmoud, Q., John Wiley and Sons, 163-187.

Graham, S., Simeonov, S., Boubez, T., Davis, D., Daniels, G., Nakamura, Y. & Neyama, R. (2002). Building Web Services with Java. SAMS.

Hausmann, J. H., Heckel, R., & Lohmann, M. (2004). Model-based discovery of Web Services. *Proceedings of International Conference on Web Services (ICWS 2004).*

Higuera, C. d. l. (2000). Current trends in grammatical inference. *Proceedings of Joint IAPR Int. Workshops SSPR & SPR 2000.*

ISIS.(2001). GME 2000 User's Manual, Version 2.0. Vanderbilt University.

Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T. (2003). Model-integrated development of embedded software. IEEE. 91(1), 145-164.

Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing domain-specific design environments. IEEE Computer, 34(11), 44-51.

Lédeczi, Á., Davis, J., Neema, S., Agrawal, A. (2003). Modeling methodology for Integrated simulation of embedded systems, ACM Transactions on Modeling and Computer Simulation. 13(1), 82-103.

Lopes, D., & Hammoudi, S. (2003). Web service in the context of MDA. *Proceedings of International Conference on Web Services (ICWS'03).*

Mantell, K. (2003). From UML to BPEL: model driven architecture in a Web Services world. http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel/.

Olson, A. M., Raje, R. R., Bryant, B. R., Burt, C. C., & Auguston, M. (2004). UniFrame-a unified framework for developing service-oriented, component-based, distributed software systems. Service-Oriented Software System Engineering: Challenges and Practices, ed. Stojanovic, Z. and Dahanayake, A., Idea Group, 68-87.

Raje, R. R., Auguston, M, Bryant, B. R., Olson, A. M., Burt, & C. C. (2002). A quality of service-based framework for creating distributed heterogeneous software components. Concurrency and Computation: Practice and Experience, 14(12), 1009-1034.

Sivashanmugam, K., Verma, K., Sheth, A., & Miller, J. (2003). Adding Semantics to Web Services Standards. *Proceedings of International Conference on Web Services (ICWS'03).*

Zhao, W., Bryant, B. R., Burt , C. C., Gray, J. G., Raje, R. R., Olson, A. M., & Auguston, M.(2003). A generative and model driven framework for automated software product generation. *Proceedings of CBSE 6, the 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction.*

[i] http://ws.apache.org/axis/
[ii] UML^{TM} - Unified Modeling Language - http://www.omg.org/uml
[iii] JVLC - Journal of Visual Languages and Computing-http://www.elsevier. com/locate/jvlc
[iv] Interview with Keith Short, http://www.theserverside.net/talks/ library.tss#KeithShort
[v] Note that the ER model is not intended to replace the existing modeling language such as UML or Petri Nets – those modeling languages have their own advanced features for a specific domain to model. Here the ER model is chosen as an intermediate form only for exchanging models of a close type or serving a close purpose but with variant notations across different modeling tools and environments.
[vi] http://bit.csc.lsu.edu/~chen/chen.html
[vii] Meta-Object Facility - http://www.omg.org/technology/documents/formal/mof.htm
[viii] RMI - Remote Method Invocation: http://java.sun.com/products/jdk/rmi/index.jsp
[ix] J2EE - Java 2 Enterprise Edition: http://java.sun.com/j2ee/
[x] CORBA® - Common Object Request Broker Architecture: http://www.omg.org/corba/
[xi] http://www-3.ibm.com/software/ad/library/standards/ocl.html
[xii] MDA - Model-Driven Architecture - http://www.omg.org/mda
[xiii] OMG - Object Management Group -http://www.omg.org/
[xiv] http://www.omg.org/cgi-bin/doc?lt/2003-11-4
[xv] http://codip.grci.com/odm/draft/submission_text/ODMPrelimSubAug04R1.pdf
[xvi] XMI - XML Metadata Interchange - http://www.omg.org/technology/ documents/formal/xmi.htm
[xvii] BPEL4WS - Business Process Execution Language for Web Services - http://www-128.ibm.com/ developerworks/library/specification/ws-bpel

A Non-Invasive Approach to Assertive and Autonomous Dynamic Component Composition in Service-Oriented Paradigm

Fei Cao
(University of Alabama at Birmingham, Birmingham, AL, USA
caof@cis.uab.edu)

Barrett R. Bryant
(University of Alabama at Birmingham, Birmingham, AL, USA
bryant@cis.uab.edu)

Rajeev R. Raje
(Indiana University Purdue University, Indianapolis, IN, USA
rraje@cs.iupui.edu)

Andrew M. Olson
(Indiana University Purdue University, Indianapolis, IN, USA
aolson@cs.iupui.edu)

Mikhail Auguston
(Naval Postgraduate School, Monterey, CA, USA
auguston@cs.nps.navy.mil)

Wei Zhao
(University of Alabama at Birmingham, Birmingham, AL, USA
zhaow@cis.uab.edu)

Carol C. Burt
(University of Alabama at Birmingham, Birmingham, AL, USA
cburt@cis.uab.edu)

Abstract: Component-based software composition offers a development paradigm with reduced time-to-market and cost while achieving enhanced productivity, quality and maintainability. Existent work on the composition paradigm are of a static composition paradigm, which is not sufficient in distributed environment, in which both constituent components and assembled distributed system are subject to dynamic adaptation. This paper presents two types of dynamic composition for distributed components: assertive and autonomous over .NET based Web Services environment. Two case studies are provided: the first one illustrates at a low level how the underlying infrastructure enables the dynamic composition; the second one illustrates at a high level how dynamic compositions are specified.

Keywords: dynamic component composition, Service Oriented Architecture, Web Services, assertive composition, autonomous composition, intermediate code manipulation, aspect-oriented programming, aspect weaving, .NET, Common Language Runtime
Categories: D.2.3, D.2.12, D.2.13, D.2.7, D.3.3, H.3.5, I.2.8

1 Introduction

With the increasing demand for scalability, reasonability and correctness of software systems, software development has evolved into a process of composing existing software components, as opposed to constructing a new software system completely from scratch [Heineman, 01]. Economically, by reducing time-to-market, this approach has improved the economic and productivity factors of software production [Devanbu, 96]; Technically, by separating overall functionality into small units, component-based software development also offers a means for better manageability [Brown, 00] and predictability [Hissam, 03] of the constructed software system.

Features of Distributed Components

With the advancement of internet technology, component-based software development has unleashed its impact into the distributed environment, while exhibiting such new features as follows:

a. The scope of component selection and reuse is extended. Consequently, component composition requires a prerequisite discovery process for identifying a matching component.
b. Distributed components are usually heterogeneous with respect to implementation languages, and host platforms. With different type systems or component models, interoperation between components will not be possible without leveraging proper bridging technology.
c. Because of the unpredictability of network transport, not only functional properties, but also non-functional properties (e.g., Quality of Service [Raje, 02], economical properties such as pricing of service) are of critical concern to guarantee the proper delivery of services offered by the assembled distributed software systems. QoS includes availability, throughput, and access control, to name a few.
d. The coupling between components is loose. A deployed component in a distributed system is subject to frequent adaptation[1] or replacement with a new version to accommodate ever-changing business requirements externally as well as the computing resource status internally. Those requirements can be either functional or non-functional.

WS as a New Paradigm for Distributed Component Composition

Those new features pose new problems for developing software systems based on distributed components. Recent years have seen the emergence of Web Services (WS) technology as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA) [Colan, 04], the open standard description language XML and transportation protocol HTTP. Consequently, distributed component composition can be achieved by wrapping heterogeneous components with a WS layer for interoperation. Using WS as a common communication vehicle, component interoperation is greatly simplified

[1] Here adaptation is defined as component compositon and decomposition; component composition and decomposition are the means to realize adaptation.

compared with such bridging technology as CORBA[2], where different interoperation implementations are needed for each pair of components contingent on their underlying implementation technologies. In the remaining part of this paper, the term component in a distributed environment is equivalent to a WS: we use it to correlate the canonical concept of a software component [Szyperski, 02].

The Need of a Dynamic Component Composition Paradigm in WS

In addition to offering an interoperability infrastructure for distributed components, WS also incorporates service discovery infrastructure in accordance with SOA. With problem (a) and (b) being embraced, current WS technology is yet to address the concerns as set forth in (c) and (d). Specifically,

1). in mission critical scenarios such as finance or military, there is a need for *guarantee of service availability* continuously, rather than shutting down the system for services adaptation;
2). in distributed environments, service consumption experiences are dynamic and desirable to be seamless, thus the customizability of service dynamically is of vital importance in a service-oriented environment.

As such, static component composition is not adequate, and both functional and non-functional property adaptation need to be applied in a dynamic fashion. Along this line, this paper presents a dynamic component composition paradigm in WS environment for adapting WS functionally and non-functionally while maintaining the availability of WS.

This paper presents a dynamic component composition paradigm based on the .NET Common Language Runtime (CLR) [Gough, 02]. We chose .NET because it is a thorough, fundamental re-architecting of a distributed computing platform based on WS, while other application server support for Web Services tend to be designed more as another client, or presentation tier for the back-end systems, with communication tier based on RMI or RMI/IIOP rather than a strictly XML protocol based such as .NET [Newcomer, 02].

This paper is organized as follows: Section 2 describes background information. Section 3 provides an overview of the approach, as well as salient features. Section 4 describes design and implementation. Section 5 provides two case studies. Section 6 provides the benchmarking for the approach. Section 7 describes related work. We conclude in Section 8 together with the description of future work.

2 Background

The .NET framework is a platform for software integration, with Common Language Runtime (CLR) for integrating software at the single operating system process scale, and with XML WS for integration at the internet scale. CLR is the .NET equivalent to the Java virtual machine, but offers more features such as using Common Intermediate Language (CIL) based on the Common Type System (CTS) [Gough, 02] to translate .NET languages before execution, thereby offering cross-language interoperability for .NET languages based on CIL. The code to be translated into CIL

[2] CORBA® - Common Object Request Broker Architecture: http://www.omg.org/corba/

and then to be executed by CLR is also called *managed code*. Also CIL includes rich metadata information for describing software module contracts to achieve *managed execution*, with the benefits of security and scalability.

The scope of this paper is not to at provide a full description of .NET CLR and XML WS, but rather to present our approach on capturing WS at the CLR level, then applying in-memory CIL code manipulation at runtime to realize dynamic component composition.

3 Overview of the Approach

3.1 Runtime Code Manipulation Through Assertive and Autonomous Composition Rules

Figure 1 provides an overview of the dynamic composition approach. In the left pane of the *execution unit*, the .NET XML WS, which is specified with Web Service Description Language (WSDL), is a layer built on top of .NET applications (1), which in turn runs over CLR (2). Consequently, .NET based XML WS can leverage the benefits of managed execution, where the .NET application is captured in the form of CIL (2), which is to be Just-In-Time (JIT) compiled into native code and executed (3). Therefore, by manipulating CIL derived from the XML WS implementation language, WS components can be composed at runtime.

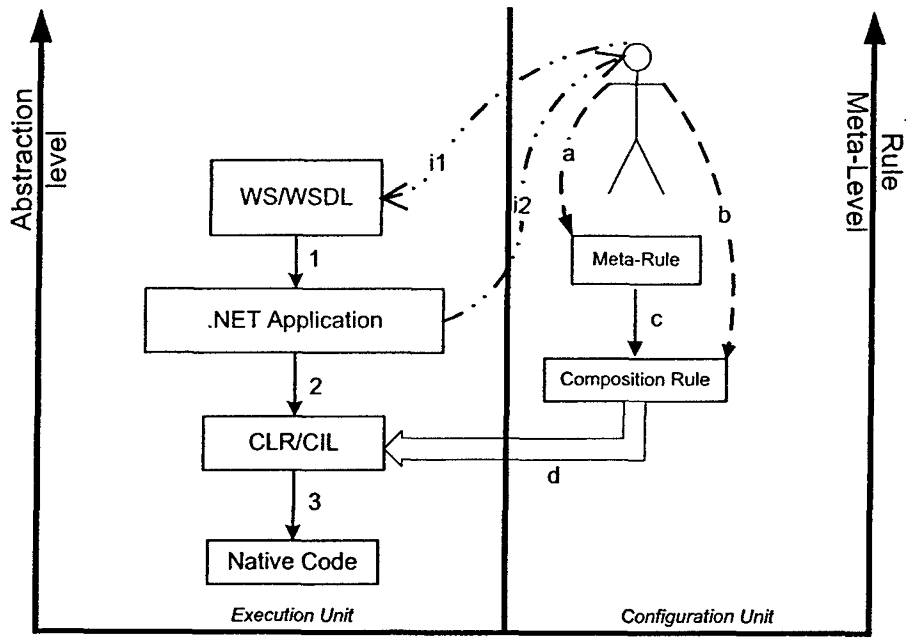


Figure 1: Overview of the dynamic composition approach

The manipulation of CIL is illustrated in the right pane of the *configuration unit*, which is comprised of a stack of composition rules with a meta-level hierarchy. Composition rules are specifications for component composition (d). Meta-rules are specifications of triggering conditions for applying the composition rules, and the firing of the composition rules is enabled through a rule execution engine automatically (c). The use of rule engine for applying composition rules is useful for implementing *autonomous compositions* based on the runtime status quo. The actor icon represents a configuration console in a manual manner for both meta-rules (a) and composition rules (b). While the composition enabled through path (a->c->d) represents autonomous composition, the composition path of (b->d) represents the *assertive composition.* The configuration decision is based on WSDL exposed by WS (i1); WS itself can in turn assume the configuration role for specifying component composition reactively (i2).

3.2 Salient Features

The dynamic component composition approach also includes the following salient features:

1). Non-invasive

▪ *Non-invasive to application code for separation of composition concerns* The WS composition is realized through in-memory IL manipulation as opposed to off-line invasive code change. The non-invasive change is often desirable as a WS vendor may deliver the software package in binary form. Also even though it is possible to derive CIL from a .NET executable using some de-compilation tools, invasively changing either original source code or derived CIL code will require unloading, recompiling and redeployment of the original WS application, which compromises the availability of WS. Moreover, the invasive change of WS code will pollute the original application such that recovering it will become difficult, which introduces the common version control problems for software systems.

▪ *Non-invasive to platform for portability.* The composition through manipulation of CIL at runtime (Figure 1-d) requires the interception of the managed execution. Instead of re-implementing the CLR such as rewriting open source CLR Rotor [Stutz, 03] to invasively add a listener for execution interception at the compromise of portability of CLR, we use a pluggable, configurable CLR profiling interface to achieve this goal, which can be enabled and disabled based on composition needs with ease to reduce unnecessary overhead.

2). Language neutral for cross-language component composition

By specifying composition rules based on WSDL, which in turn is based on alanguage neutral XML schema[3], and code manipulation at the intermediate code (CIL) level based on language neutral CTS, WS components implemented in different .NET languages can be composed across language boundaries.

3). Adaptable composition.

[3] http://www.w3c.org/2001/XMLSchema

With the configuration unit as a separate entity applied to runtime as shown in Figure 1, not only is the composition concern separated, but also it can be updated to realize adaptable composition at runtime.

The following section presents in detail the design and implementation of the dynamic component composition in Peer-to-Peer (P2P) scenarios, particularly, how the composition rules are specified to facilitate assertive and autonomous configuration.

4. The Design and Implementation of Dynamic Component Composition

4.1 Peer-to-Peer (P2P) Component Composition

Figure 2 illustrates the architecture for the dynamic component composition based on .NET WS environment. In our work, each component is hosted in an infrastructure *DynaCom*, which is essentially a profiler-enabled CLR to be detailed in Section 4.2. DynaCom is used as a proxy to for components to interoperate with components in other locations through WS. Meanwhile, DynaCom can intercept the execution of the hosting components and change the behaviour of the executing components dynamically. DynaCom is based on our prior work on using a profiling approach for dynamic service provisioning [Cao-a, 05], but here it is tailored to component composition.

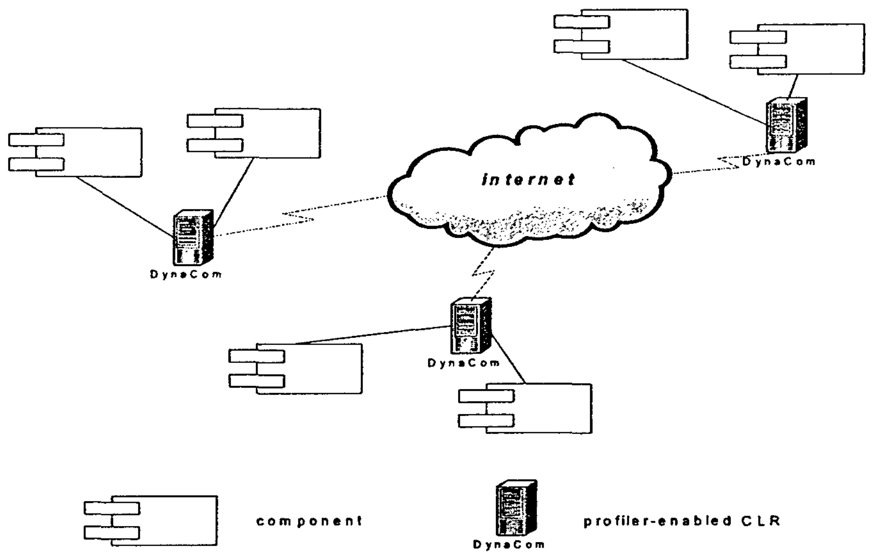


Figure 2: The P2P component compositions in .NET WS environment

The component composition model shown in Figure 2 represents a P2P paradigm, which is the primary composition model to be addressed in this paper. This choice is based on the observations that P2P and dynamic composition are tightly associated:

1). *P2P as an agile mode to accommodate dynamic features.* While WS orchestration by executing BPEL4WS[4] in the execution engine represents a centralized composition model, it has been observed that such a composition model compromises scalability, availability, and security for the server [Chen, 01]. With the highly dynamic features in distributed environment, P2P component composition paradigm will be more widely used.

2). *Dynamic composition is the necessary means for realizing P2P computation in a distributed environment.* While component composition usually requires the generation of glue/wrapper code [Cao, 02], the physical location for hosting the generated glue/wrapper code is a hard problem in P2P mode without central management and storage units. Dynamic composition, with glue/wrapper code generated in memory and JIT compiled and executed at runtime, provides a solution for P2P component composition without the physical code placement issues.

4.2 DynaCom Exposed

Figure 3 provides an anatomy of DynaCom. The part enclosed by the big square represents the enabling mechanism for dynamic composition, which is transparent to the components to be composed above the big square.

Our work is built upon the ASP.NET[5], a WS implementation package based on the .NET framework. In ASP.NET, Internet Information Service (IIS)[6] is used to accept the incoming WS SOAP (Simple Object Access Protocol) [Newcomer, 02] message transported over HTTP (1). Upon acceptance of the WS request encoded as a SOAP message, an IIS filter will launch a work process (aspnet_wp.exe), which in turn will launch CLR (2) to run the WS application in the mode of managed execution. At this point, the WS application is rendered as in CIL subject to be JITcompiled into native code and executed (6). In order to adapt WS, it is needed to intercept the WS call at the CIL level before it is compiled. While it is reasonable to implement the expected functionalities in the CLR open source of millions of lines of code such as Rotor [Stutz, 03], we feel it too expensive an effort. Instead, we use the CLR profiling API to implement a *Profiler* as event handlers, and register them as listeners for the events generated from the CLR (3). In contrast to the conventional publisher/listener model, which is often of a client-server relationship, the profiler here will be mapped into the same address space for the profiled application as an in-process server.

[4] BPEL4WS - Business Process Execution Language for Web Services - http://www-128.ibm.com/developerworks /library /specification/ws-bpel
[5] http://asp.net
[6] http://www.microsoft.com/WindowsServer2003/iis/default.mspx

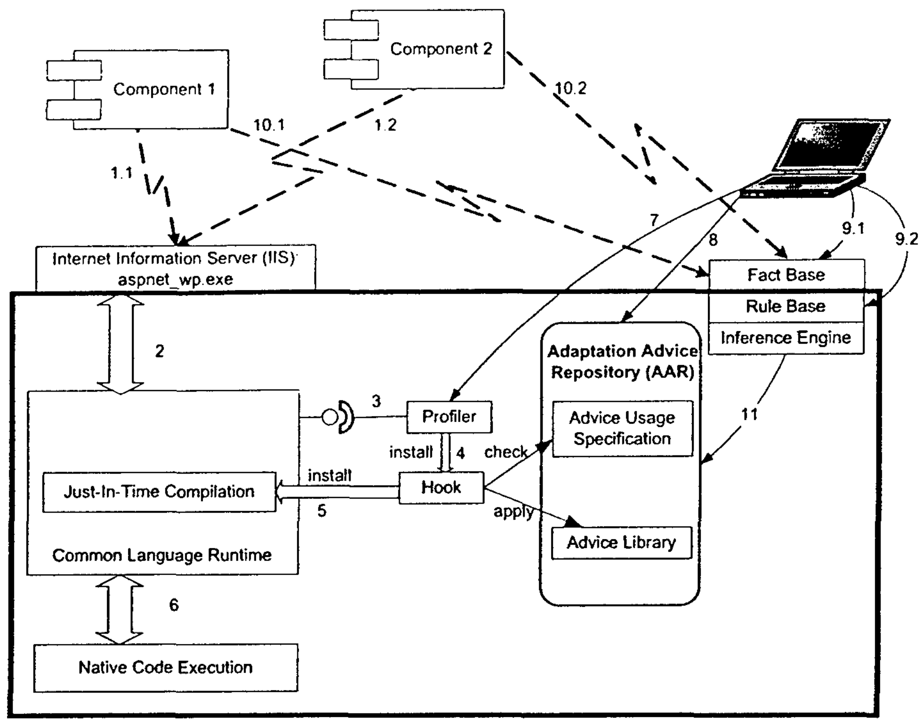


Figure 3: The Architecture of ***DynaCom: Dynamic Component*** *composition enabling unit, which includes the part enclosed by a bold-border rectangular and the IIS, facts. The parts of IIS and facts are accessible to the remote components, while the enclosed part of DynaCom are only accessible locally. The dashed line of 1 and 10 represents remote access, while all the remaining solid lines represent local access. The laptop icon represents the local configuration unit to DynaCom.*

The events generated from the CLR are the result of managed execution, including but not limited to garbage collection, class loading/unloading, CLR startup/shutdown and JIT compilation. The event of our interest is JIT compilation, for which we implement in-memory CIL manipulation for the event handler. The adapted CIL will then be JIT compiled and executed resulting in changed WS behavior. A one-shot change to CIL will reduce the traceability of adaptation, impede the removal of the imposed adaptation (thus incapable of dynamic decomposition), and restrict the flexibility of further adaptation. Therefore, we interpose a *Hook* code (4,5) in the WS application to be adapted, which will check the *Adaptation Advice Repository* (AAR) for applicable adaptation advice. The term "advice" is further explained in the next section. AAR is located in a shared memory for fast access during in-memory CIL manipulation. The AAR includes an Advice Library storing predefined reusable advice in the compiled managed code form, as well as an *Aspect Usage Specification* (AUS) component to indicate applicable advice for WS. The Profiler and the AAR are subject to external configuration (7-11): for 7, the configuration is used to narrow down the scope of profiling; for 8-11, the configuration is used to dynamically specify adaptation rules, among which 8 corresponds to a direct manipulation of adaptation rules, while 9-11 corresponds to indirect manipulation of adaptation rules through a rule inference engine. The inference engine can dynamically inject AUS into AAR based on the rule specification, which is to be detailed in Section 5.2. The laptop icon in the upper-right corner represents the local configuration unit. The configuration unit for DynaCom can adopt a GUI interface or an API interface. In our work, we use a simple console for the local configuration unit handling configuration 7-9, while configuration 10-11 is realized through an API interface.

4.3 Dynamic Component Composition Through Dynamic Aspect Weaving

4.3.1 Modularized Component Composition

In Figure 2, each DynaCom only hosts 2 components, which is for simplicity purpose in illustration. In reality, a DynaCom may be hosting multiple components. Consequently, a component handling a crosscutting concern may be expected to be composed with multiple other components. Thereafter, it is not possible to specify adaptation for every individual component upon changing of requirements. Instead, there needs to be a means to abstract the adaptation in a modularized way. Aspect-Oriented Programming (AOP) [Kiczales, 97] offers a means to abstract cross-cutting concerns in a modularized way called an *aspect*, and the concerns can be weaved using weaver technology into the base program based on the *join point model*, which specifies the destination to weave concerns. In the same vein, we specify the adaptation advice in the AAR in a modularized way following AOP style[7]. To weave and unweave the specified advice, we instrument the hooks at both the entry (*pre-hook*) and exit point (*post-hook*) of the WS method to be adapted, which are used to check into the AAR to see if corresponding *before advice* and *after advice* is applicable: the former performing some pre-processing before the actual WS method execution, while the latter performs some post-processing immediately before the WS method execution returns. Such pre- and post- processing capacity can be used to instrument codes for addressing non-functional concerns, such as applying access control upon the entry into the WS method, or applying state persistency service for the executed WS application upon the end of the WS call. Also included in the pre-hook are the instructions to check if an *around advice* is specified or not, and a jump instruction to redirect the execution to the exit point of the WS application. The jump instruction is to be activated if an around advice is found valid in the AAR. With around advice, the original WS will be replaced with new behaviour specified in that around advice. Consequently, not only the original WS can be decorated, it can also be overridden completely, which is necessary when a buggy WS is identified and

[7] AOP also offers a means for separating composition specification from components to be composed, with the underlying weaver realize the composition. As such, in case the components to be composed do not involve crosscutting concerns, the component composition is still specified in the same way as an aspect weaving specification with AUS.

needs to be removed, or an old service module need to be updated. The around advice sufficiently offers a delegation and wrapping approach for component composition which is exemplified in Section 5. By using a hook for weaving, advice can be applied dynamically and proactively. Meanwhile, unweaving advice can be realized by dis-activation of the corresponding AUS in AAR. Figure 4 is the CIL manipulation template for adapting a WS method.

```
IL_0000: ldstr "classname/method_name/parameter_name_list/returntype/before"      ┐
IL_0005: call    void dynaweave.hook::advising(string) //to check & apply before-advice │
IL_000a: pop   //to maintain the original stack                                  │
IL_000b: ldstr "classname/method_name/parameter_name_list/returntype/around"      │   pre-hook
IL_0010: call  void dynaweave.hook::advising(string) //to check & apply around-advice │
IL_0015: brtrue IL_020b                                                          ┘
IL_001a:  <Original Method body in IL>
............
IL_0200: ldstr "classname/method_name/parameter_name_list/returntype/after"       ┐
IL_0205: call   bool dynaweave.hooker::advising(string) //to check & apply after-advice │   post-hook
IL_020a: pop //to recover the original stack after original method is executed    ┘
IL_020b: return
```

Figure 4: Instrumentation of IL code of a WS method

4.3.2 Specifying Component Composition via Aspect Usage Specification

The AOP weaving specification in AspectJ [Kiczales, 01] can be adapted for component composition specification in terms of aspect weaving as illustrated in Table 1.

Component Composition		Aspect Weaving Specification
Sequential	a precedes b	after (a) {b; }
	a follows b	before (a) {b; }
Wrapping	a is wrapped by b at the beginning and c at the end	around (a) {b; proceed(); c; }
Overiding	a is overriden by b	around (a) {b; }

Table 1: Composition specification in the form of aspect weaving

The aspect weaving specification is represented in AUS. The type system used in the AUS in AAR can be based on the object-oriented CTS of CIL, for which each CLR hosted language is translated to before being JIT compiled. Therefore, such specification is applicable to all WS applications running in CLR, which provides a language-neutral way for AUS. However, writing adaptation AUS based on low level CTS is error-prone and not necessary for high-level AUS. As a result, AUS is written in XML rather than in CTS, which is based on the following observations:

1) **Necessity**

- Components delivered may be in binary form with source code being unavailable, thus AUS at the application code level is not feasible. On the other hand, components in the .NET WS environment are exposed through the WSDL interface, which offers a reference point for specifying WS component adaptation.
- AUS, as the specification reflecting the business requirement adjustment (by composing and decomposing related components), should have an abstraction level close to business requirements, rather than being tied to underlying implementation details.
- XML-based specification for AUS can be directly serialized and queried by hooks using XML manipulation APIs such as DOM or SAX or XQuery[8].

2) **Sufficiency**

- Web Service Description Language (WSDL) is based on the XML Schema, which is another language neutral type system that can be mapped to the language-neutral CTS. The XML Schema based specification is parsed and translated to CTS to be matched against the string provided by the hook such as described in IL_0000, IL_000b, IL_0200 in Figure 4. The AUS in AAR accords with XML schema as illustrated in Figure 5.

```
<wsdl:operation name="apply_advice">
  <wsdl:input message="tns:advicetype"/>
  <wsdl:input message="tns:return_type"/>
  <wsdl:input message="tns:classname"/>
  <wsdl:input message="tns:methodname"/>
  <wsdl:input message="tns:parameter_list"/>
  <wsdl:input message="tns:advicename"/>
</wsdl:operation>
```

Figure 5: The AUS schema

Associated with each *advicename* is the path information for actual advice in the form of managed code stored in the AAR. All the advice code is defined as a template with the tuple *<Classname, Methodname, Parameter_List>* as parameters, which offers reusability of advice. Such advice can be pre-built in any .NET language and compiled into managed code. If a matching advice is found, then the advice code will be loaded from the corresponding path and called. In our work, the wild-card characters are also supported for AUS.

[8] http://www.w3c.org

4.3.3 Autonomous Component Composition Using Rule Inference Engine

Functionality for the composed distributed software systems can be predicted based on the constituent components [Hissam, 03], thus a component composition based on functional requirements can be specified assertively. In contrast, non-functional properties such as pricing based on end-to-end delay (service consumption duration) for composed distributed software systems can only be reasoned about at runtime because of their dynamic characteristics. As such, a distributed software system needs to self-adapt itself by composing and decomposing components autonomously to achieve the expected QoS. The self-adaptation decisions can be collectively built into a knowledge base proactively and retroactively. Therefore, the complete dynamic component specification in terms of dynamic, autonomous aspect weaving takes the following rule:

```
apply [aspect_name] when [logical_condition]
```

The *when* clause represents the condition under which the action *apply [aspect_name]* need to be performed. Consequently, the AUS schema in Figure 5 will be augmented with an attribute *when* for the *wsdl:operation* element. The use of rule inference provides a means for not only separation of concerns between business rules and the underlying technical implementations for component compositions, but also autonomous composition at runtime.

In our work, we use Jess [Friedman-Hill, 05] as the underlying inference engine, which is a forward and backward chaining rule engine for the Java platform. Associated with the inference engine are the fact bases and the rule base as shown in Figure 3. The rule base is only accessible to the local hosting site, and represents local autonomous composition policies; comparatively, the fact base is exposed to both the local and remote site, which can be manipulated by either the local configuration unit, local components, or remote components. The fact bases of different DynaCom are federated, and a local rule engine can query remote fact base for triggering an action. This is useful when a local composition rule is dependent on remote component status (which is reflected in the remote fact base). For example, the unavailability of a remote components during a certain period of time will trigger the local component to connect to an alternative component, which offers a means of fault tolerance.

Jess offers a hybrid programming paradigm between the Java language and declarative rule specification: the Java code can invoke the Jess rule engine while the Jess rules invoke Java code. In order for Jess fact base to interoperate with remote components, as well as to enable the Java-based inference engine interoperable with .NET environment, we wrap the Java-based Jess API with a WS layer using Java WSDP[9].

[9] Java WSDP – Java Web Services Developer Pack – http://java.sun.com/webservices/jwsdp/index.jsp

5 Case Study

In this section we present two case studies. The first one is an assertive dynamic composition example which is also intended to illustrate how every part shown in Figure 3 works together. The second one showcases a dynamic composition paradigm of autonomous composition.

5.1 Composition Crosscutting Credit Authorization Components

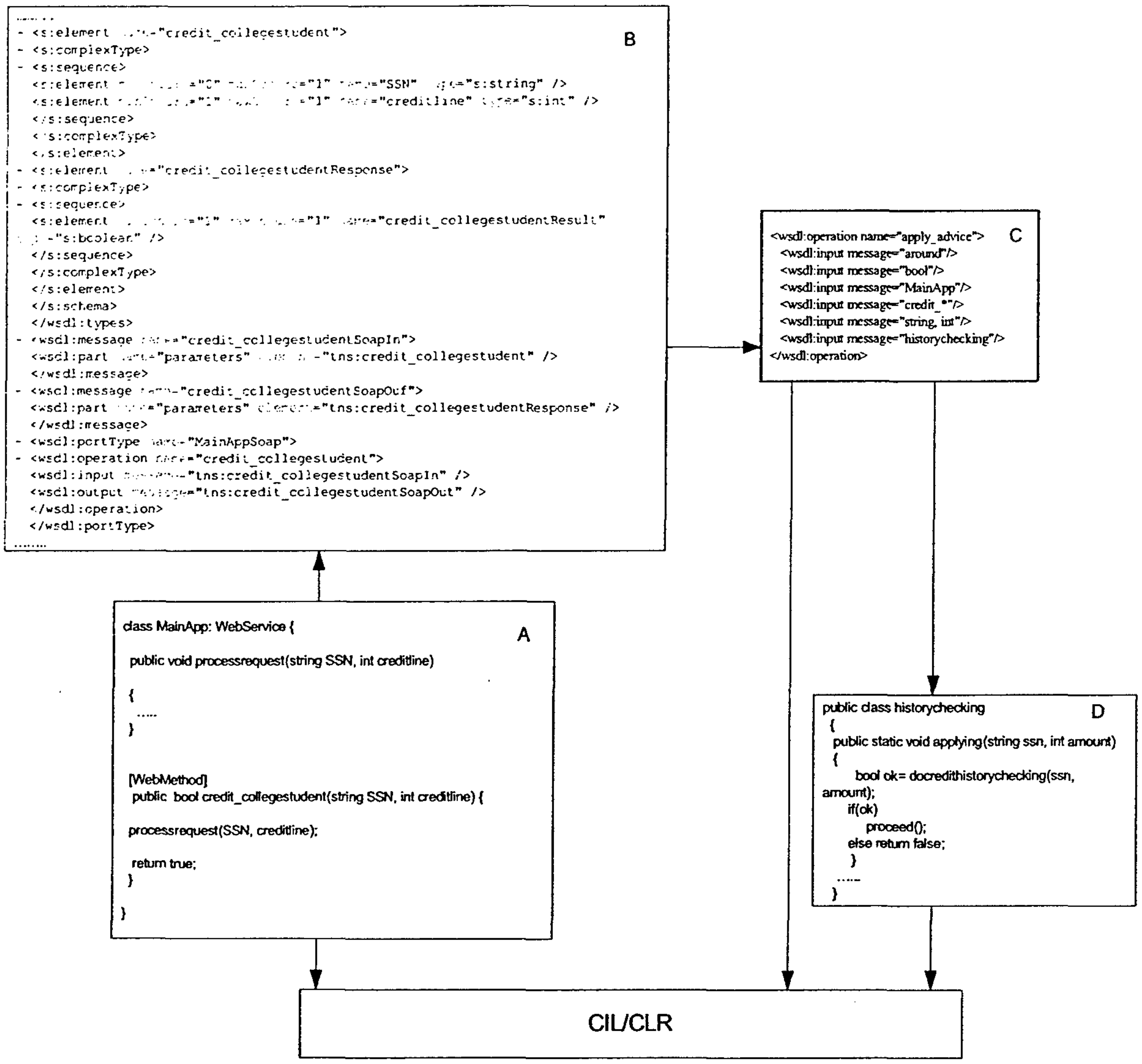
Figure 6 provides an example of a college student credit authorization WS to demonstrate the assertive dynamic component composition for a non-functional concern: access control. Figure 6-A provides a simple WS application written in C#, which provides a WS method for authorizing credit card application based on the Social Security Number (SSN[10]) and the expected credit line. The corresponding WSDL in Figure 6-B can be automatically generated from the source code in Figure 6-A based in ASP.NET, which in turn is to be exported and used as the basis for AUS as well. Figure 6-C is an AUS with an around advice to apply credit history checking before any credit card application request is processed. The AUS represents a sequential composition specification for a component encapsulating crosscutting concerns (here *HistoryChecking*). The wild card specification in credit_* represents all credit application with the request name preceded with "credit_". Figure 6-D is the source code for the pre-built credit history checking advice, which can be written in any .NET language (here C#) and is compiled and persisted in the managed code form. The type systems in Figure 6-A, Figure 6-C, Figure 6-D are translated into CIL and matched up in CLR. Once a match holds, the advice in Figure 6-D will be called by the hook instrumented at runtime. The WS application source code level detail is transparent to AUS in Figure 6-C, as well as to the HistoryChecking component in Figure 6-D. By instrumentation of intermediate code, component composition can be realized across language boundaries without invasively changing application source code.

5.2 Composing Travel Planning Components

The former section demonstrates how each part in DynaCom is integrated together for assertive dynamic component composition, particularly how the intermediate code manipulation enables the component composition across language boundaries without invasively accessing the application source code. This section will further explore the dynamic composition for multiple components for travel planning, which not only includes assertive dynamic composition, but also autonomous dynamic composition using the Jess rule inference engine. Complementing the previous case, this case focus on the user level component composition specification as opposed to dwelling on the low level intermediate code manipulation.

In Figure 7, the boxed part contains the WS components for travel planning, with those above the box representing the types used in the WS components. Each customer plans the travel through a travel agent *Travel_Agent (TA)*. The travel agent

[10] An identification number used to identify income earners in the United States.



```
......
- <s:element name="credit_collegestudent">
- <s:complexType>
- <s:sequence>
  <s:element [illegible]="0" [illegible]="1" name="SSN" type="s:string" />
  <s:element [illegible]="1" [illegible]="1" name="creditline" type="s:int" />
  </s:sequence>
  </s:complexType>
  </s:element>
- <s:element name="credit_collegestudentResponse">
- <s:complexType>
- <s:sequence>
  <s:element [illegible]="1" [illegible]="1" name="credit_collegestudentResult"
type="s:boolean" />
  </s:sequence>
  </s:complexType>
  </s:element>
  </s:schema>
  </wsdl:types>
- <wsdl:message name="credit_collegestudentSoapIn">
  <wsdl:part name="parameters" element="tns:credit_collegestudent" />
  </wsdl:message>
- <wsdl:message name="credit_collegestudentSoapOuf">
  <wsdl:part name="parameters" element="tns:credit_collegestudentResponse" />
  </wsdl:message>
- <wsdl:portType name="MainAppSoap">
- <wsdl:operation name="credit_collegestudent">
  <wsdl:input message="tns:credit_collegestudentSoapIn" />
  <wsdl:output message="tns:credit_collegestudentSoapOut" />
  </wsdl:operation>
  </wsdl:portType>
........
```

```
<wsdl:operation name="apply_advice">
  <wsdl:input message="around"/>
  <wsdl:input message="bool"/>
  <wsdl:input message="MainApp"/>
  <wsdl:input message="credit_*"/>
  <wsdl:input message="string, int"/>
  <wsdl:input message="historychecking"/>
</wsdl:operation>
```

```
class MainApp: WebService {

 public void processrequest(string SSN, int creditline)

 {
   .....
 }


 [WebMethod]
  public bool credit_collegestudent(string SSN, int creditline) {

 processrequest(SSN, creditline);

  return true;
 }

}
```

```
public class historychecking
  {
   public static void applying(string ssn, int amount)
   {
        bool ok= docredithistorychecking(ssn,
amount);
      if(ok)
           proceed();
      else return false;
      }
   .....
  }
```

Figure 6: Composing credit authorization component assertively(A+D)

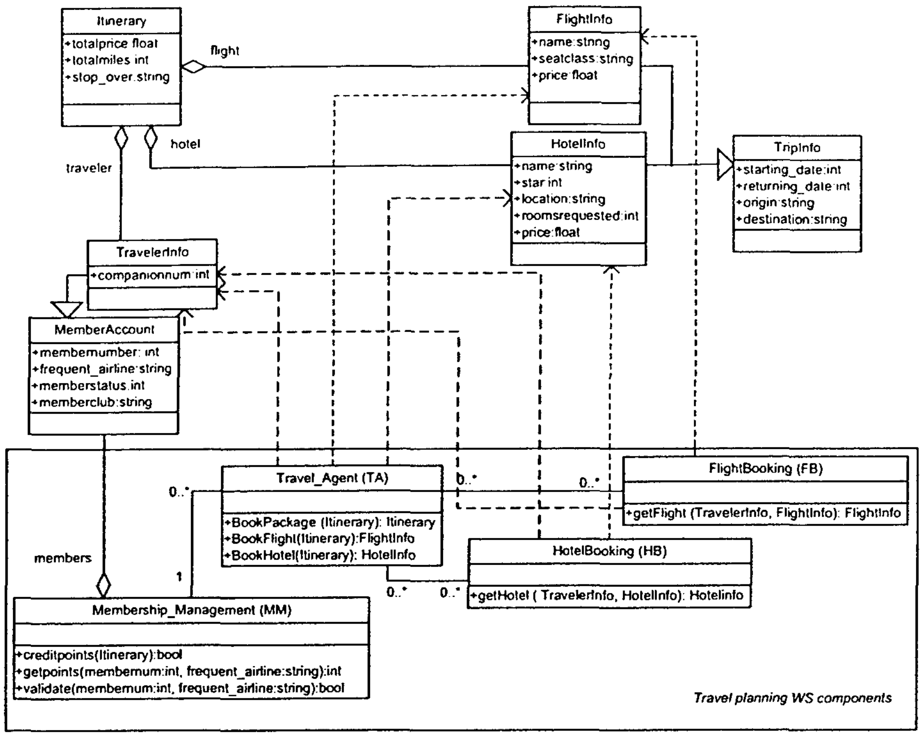


Figure 7: Class diagram for travel planning WS components

will handle both the booking of flight, *FlightBooking (FB)* and hotel, *HotelBooking (HB)*. Every traveler can credit his mileage into his own frequent flier number through *Membership_Management (MM)*. He can book the travel package including both hotel and flight, or just book one of them. He can also book for group travelers. The result of the travel booking process is the itinerary information (*Itinerary*), which includes the total cost of the trip. All those WS components in the box are loosely coupled and dynamically bound based on their partnership, service charge, and QoS. Figure 8 illustrates the travelling components composition process with sequence diagram. The italicized part represents the dynamically composed components; the TA and its associated methods represents the static front end travel agent components to the customers with back end components dynamically composed on demand.

5.2.1 Static Front End

During travel planning, the customer starts from TA WS method *BookPackage*, with the backend components dynamically composed to fulfill the travel planning purpose. The TA serves as front end components to the customers to be dynamically bound to backend WS components, and the *BookTravel* method is implemented as shown below:

```
Itinerary BookPackage (Itinerary it)
 {
     FlightInfo fi;
```

```
            HotelInfo hi;
            fi=BookFlight (it);
            hi=BookHotel(it);
            return combine(it1,it2);
        }
```

5.2.2 Dynamic Backend

While the front end code as shown above is static to the customer side, there are some dynamic component composition concerns in the backend that is transparent to the customers:

- Dynamic partnership

The front end TA component may have dynamic partnership with back end FB and HB (we assume membership management is centralized and statically bound in this case in accordance to the real world examples, where membership such as Social Security Account is centrally administrated by the appropriate government agency) based on their mutual contract, service charge (if the service charge is exceeding the budget, the partnership will be cancelled and a new partner will be identified), or QoS (if the service of the current partner is down, an alternative partner need to be identified). As such, the partnership should be established dynamically, which is also subject to dynamic change consequently. Figure 8 illustrates the dynamic partnership establishment by using two <<*create*>> messages before the call of *BookPackage*,

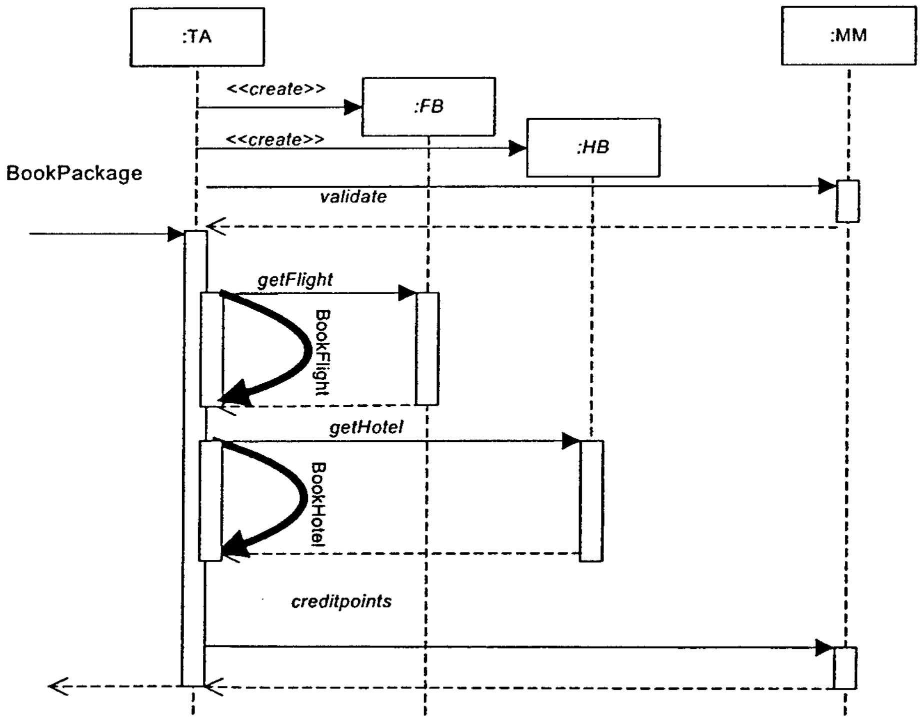


Figure 8: Dynamic composing travel planning WS components

which can be translated into the following dynamic composition specification using before advice[11].

```
   before(Itinerary *.BookPackage (Itinerary it))
   {
    this.fb=  new  FB(...);  //the  "..."part  provides  the
//information referencing the actual FB component that
//the instantiated object is bound to
    this.hb= new HB(...);
   }
```

Furthermore, the front end BookFlight and BookHotel code is dynamically overridden to delegate to the actual methods of FB and HB respectively. This is achieved using around advice as shown below:

```
   around (FlightInfo *.BookFlight (Itinerary it))
    {
     return fb.getFlight (it.traveler, it.flight);
     }

   around (HotelInfo *.BookHotel (Itinerary it))
    {
      return fb.getHotel (it.traveler, it.hotel);
        }
```

▪ Dynamic membership management

With the tightening security measures, the customer's background is subject to be checked by the central member management (MM) unit upon designated period of time. As such, a rule is added in Jess that for a given duration, the membership will be validated (e.g., background checking, passport verification) for each *BookPackage* call. Assume during the period July 1, 2005, to September 20, 2005, all traveller's membership will be validated by MM. To enable the Jess rule engine to trigger the dynamic composition of validation behavior, we need to:

1)capture the execution of *BookPackage* and relay the values into Jess fact bases;
2)have a bridge from Jess to .NET for rules to directly manipulate AAR in Figure 3.

As is mentioned in Section 4.3.3, we use WS to wrap a Java class, which in turn can interoperate with Jess. Thus, a .NET based WS component can interoperate with Jess rules. Specifically, to achieve 1), we add into the "before advice" for BookPackage the following code:

[11] For illustrative purpose, we use the syntax resembling AspectJ to specify the component composition, which in turn will be translated into XML representation as described in Section 4.3.2.

```
before(Itinerary *.BookPackage (Itinerary it))
{
  ......
  ...... //above are other advice code which are ignored
     //here for clarity

WS_Jess.assert("membernumber",
                it.traveler.membernumber  );
WS_Jess.assert("airline",
               it.traveler.frequent_airline);
Date date=getdate();
WS_Jess.assert("date",date);
//the above three lines add three
//facts to the Jess fact base through WS-Jess bridge
}
```

To achieve 2), we define a Java class which is used as a relay between Jess and .NET platform, so that whenever the rule fires, AAR in .NET can be manipulated from Jess. The Java class is defined as follows:

```
class Jess_WS{
 public static void
     apply(string advicetype, String  returntype,
           String classname, String methodname,
            String parameterlist, String advicename)
     {
      ... //code to interoperate with .NET to update AAR;
     }
}
```

The parameter list is consistent with the XML elements as shown in Figure 5. The Jess rule is specified as follows, which calls into the Java class Jess_WS:

```
(bind ?aus  (new  Jess_WS)) ;;aus_wrapper  is  the  Java
         ;;wrapper for writing AUS
         ;;into  the  AAR  through  Java-WS  bridge  using
         ;;Java WSDP as ;;described in Section 4.3.3
(bind ?para (str-cat ?membernumber "/" ?airline))
         ;;the  values  of  ?membernumber  and  ?airline
         ;;are  fed  into  the  fact  base  by  the  before
         ;;advice for BookPackage
(defrule security_control
 (date ?d &:(>= ?d 20050701)&:(< ?d 20050920))
  =>(?aus   apply  "before",  "",  "TA",  "BookFlight",
   ?para, "MM.validate"))
```

The last line defines a Jess rule specifying once the booking date is between July 1, 2005 and September 20, 2005, the membership validation advice will be applied through Jess-Java-WS interoperation before the call of *.BookFlight in .NET environment. Once the condition is satisfied during runtime, the corresponding rule will be applied autonomously for dynamic composition. Furthermore, as the Jess rule exists as a separate entity for configuration from the execution logic, the composition rule can be adapted as needed at runtime as well.

Likewise, dynamic composition can be applied to credit travel points after the travel reservation, using after advice:

```
after(Itinerary *.BookPackage (Itinerary it))
{ MM.creditpoints(it);
}
```

Furthermore, dynamic composition can be applied either assertively or autonomously as shown above for other non-functional property guarantees including but not limited to budgeting (if the cost of the requested service exceeds the budget, either to choose a cheaper service or to remove subcomponents for reducing cost), and load balancing (if current load is over capacity, the service requests are to be delegated to alternative components). As those composition specifications overlap the aforementioned dynamic composition specifications in principle, details are omitted here.

6 Performance Evaluation

Using the profiler to handle all the events generated from all managed execution in CLR is expensive and will degrade system performance significantly. Therefore, we apply optimization at three levels through configuring the profiler as indicated in (7) in Figure 3:

1). As the CLR can be launched from a shell, Internet Explorer, ASP.NET, and other customizable CLR hosts for managed execution, we configure the profiler to skip profiling for all non-ASP.NET modules hosted in CLR, which can be filtered easily based on the name of the module that launches the CLR.

2). We could further trim unnecessary profilings based on class name, or CIL method. This is possible because all managed code is translated to CIL, and the CIL level information can be derived from the corresponding WSDL for the WS; this is also necessary to avoid profiling system classes and methods.

3). We mask all unnecessary events except JIT compilation events, which is needed for handling CIL manipulation.

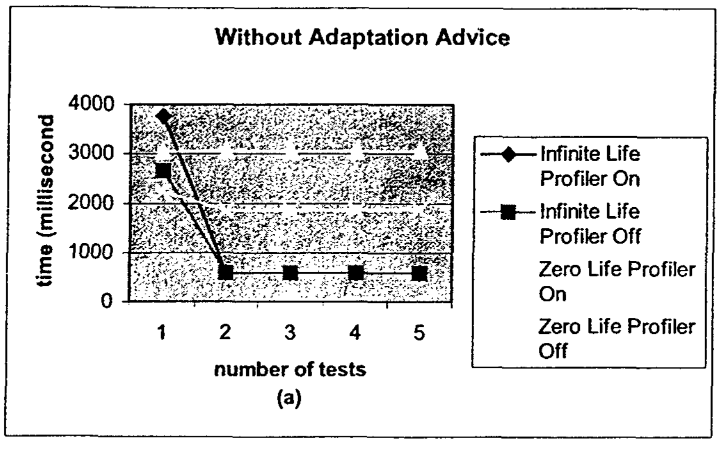
To evaluate the influence of CLR profiling-based WS adaptation on performance, we implemented a simple WS server application with 100 loops for calling a method, which contains only a single plus calculation in its body. We host this WS application on a Dell Workstation with Intel XEON CPU 2.2GHx, 1.00GB RAM, which is installed with Win XP professional version 2002 with IIS 5.1, .NET framework version 1.1.4322. We configured the profiler so that the method is to be profiled and adapted with log advice to write to a file a line of strings. A WS stub is generated by compiling the corresponding WSDL for this simple WS application. The WS stub is instrumented together with a simple client application for the client application to call the server-side WS. The client side is hosted on a Dell PC with Intel Pentium 4 CPU 1.80 GHz, 512 MB RAM which resides on the same LAN environment as the server so as to minimize the network influence during the server side performance benchmarking.

Note that the CLR profiling-based approach only applies to managed code to be loaded and JIT compiled. Therefore, we run ASP.NET in the managed mode for profiling WS to realize dynamic adaptation. ASP.NET can load one worker process to handle a pool of WS requests. Once the worker process is launched to serve the first WS requests in the pool, it continues to serve other WS requests in the same pool until the end of its lifecyle without itself being reloaded into CLR, thus it fails to profile the other WS applications in the same pool. Therefore, we adjust the setting for ASP.NET so that a new worker process will be created for each WS request so that each WS call can be captured by the Profiler and thus is adaptable. The goal of our tests is to evaluate how the adjustment of worker process lifetimes (Figure 9-a), and the enactment of profiling-based dynamic adaptation (Figure 9-b) affect the performance of WS provisioning in the peer-to-peer composition model.

For the case in Figure 9-a, we did not provide any adaptation advice when adjusting the worker process life between *zero life* (a new worker process is created for each WS request) and *infinite life* (the same worker class is used for multiple WS requests). The absence of advice execution will help clarify the influence of the changing life of a worker process on the system performance.

There are significant differences between the first call and the remaining calls for an infinite life case as the first call involves the creation of a new worker class, thus incurring more overhead than the remaining WS calls which reuse the original worker process. Also the presence of profiling does not affect performance much in the case of infinite life, as the worker process is no longer to be reloaded for new WS requests, thus the new WS will not be adapted, and the event handler in the profiling API is ignored. In comparison, the worker process with zero life will incur a performance degradation of 1.7 times slower with profiling on than with profiling off. With the absence of the profiler, the overhead incurred by adjusting from infinite life to zero life will be 3.0 times. With the absence of advice, the overall performance degradation (with profiling on, zero life for worker class) against the conventional WS provisioning scenario (with profiling off, infinite life for worker class) for this WS provisioning is 3.0*1.7=5.1. Figure 10 illustrates the performance degradation.

In Figure 9-b, we focus on evaluating the influence of active advice on the overall performance. Therefore, the worker process is set with zero life. We found the number of active advice will not affect the performance linearly, as the AUS are stored in the paging file to be shared by hooks, which constitutes a minor overhead in comparison to that incurred by hook instrumentation and calling of advice. The weaving of a matching advice in the case of zero life in Figure 9-b incurs a performance degrade of 2.2 times. Therefore, the overall performance degradation (with profiling on, zero life for worker class) against the conventional WS provisioning scenario (with profiling off, infinite life for worker class), by synthesizing the result descibed in the preceding paragraph, will be 2.2*5.1=11.2.



(a)

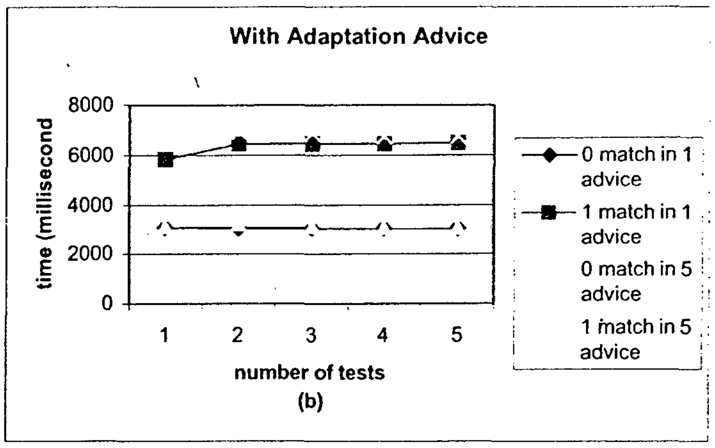


Figure 9: Benchmarking dynamic WS adaptation

In the real world deployment, we can reduce the overhead by setting the worker class to zero life at the adaptation time, then resetting it to infinite time after adaptation is done. Of course this assumes a predicable adaptation process.

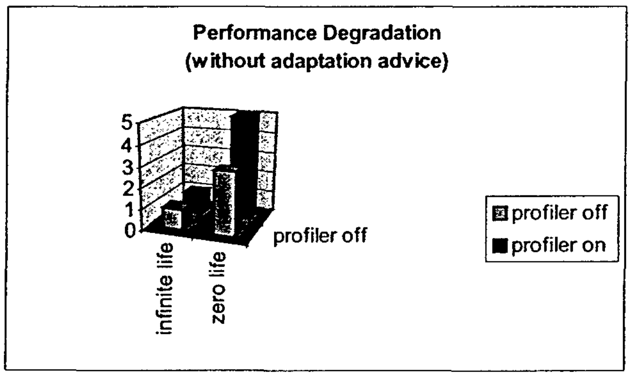


Figure 10: Performance degradation with 0 adaptation advice

7 Related Work

Component composition can be enacted at design level (e.g., [Clarke, 02], [Keller, 98]), and application code level (e.g., [Hölzle, 93], [Mezini, 98], [Seiter, 99]). In contrast, our work on component composition is enacted at intermediate code level without introducing new language constructs. With a lower-level of abstraction, our work enables cross-language component composition, while the above work restrict the component composition to a specific language. Also, none of the aforementioned work on component composition is applied at runtime, which is however necessary in distributed computing environment.

The Composition pattern has been proposed in [Clarke, 01], which uses a UML template for specifying composition of crosscutting concerns at a high level and maps sequence diagrams into AspectJ code. Our composition pattern is represented with a comprehensive framework rather than just a design-level pattern. Also a sequence diagram is used here for illustrating the dynamic partnership, with each object in the sequence diagram corresponding to a partner when mapped to dynamic composition specification. In contrast, each object in a sequence diagram ia synthesized to an aspect construct in AspectJ in [Clarke, 01]. While AOP has been applied to distributed systems for resolving crosscutting concerns ([Pulvermuller, 99], [Zhang, 03]), here we dedicate AOP to the composition purpose: for composing components handling cross-cutting concerns in a modularized way, and for separating composition from components. Moreover, we use the Jess inference engine to autonomously apply aspect weaving for component composition. While the work described in [Yang, 02] also aims at applying an aspect-oriented approach to dynamic adaptation, they only offer a means for making the AOP-based adaptation ready, without presenting any solution on how to use rule engines to trigger the adaptation. Additionally, [Duzan, 04] presents a prototype implementation in the QuO toolkit for an aspect-based approach to programming QoS-adaptive applications. In contrast, our work is targeted on loosely coupled service oriented computing as opposed to tightly coupled distributed object computing in QuO, where adaptation rules are triggered by exceptions thrown from runtime.

Our work also incorporates non-functional concerns into WS component composition. Prior work such as IBM's Web Services Level Agreement (WSLA) [Dan, 02] and HP's Web Service Management Language (WSML) [Sahai, 02] incorporate the notion at higher-level presentation, rather than address it at a lower-level platform layer. We believe a treatment at a platform layer is necessary toward thoroughly addressing non-functional concerns for WS.

Our work is rooted in the UniFrame project ([Raje, 02], [Olson, 05]), which aims at creating a framework for seamless integration of distributed heterogeneous components. In UniFrame, component composition is also following the peer-to-peer paradigm, which is enabled through a discovery services in search of a matching component. Once a searched component does not match the requirement functionally or non-functionally, the search process will be launched again, which exhibits the autonomous features similar to that described in the work presented here. While the work presented here is scoped at the service-oriented computing paradigm for component composition, the principles can be integrated into UniFrame as well.

8 Conclusion and Future Work

This paper presents a dynamic component composition approach under service-oriented paradigm in the .NET environment. By using intermediate code manipulation, component composition is 1) possible to cross language boundaries so long as they are CLR-compliant; 2) achieved in a non-invasive manner; 3) implemented not only in an assertive manner, but also in autonomous manner using a rule inference engine; 4) specified using the AOP paradigm for separating composition specification from components to be composed, and for modularized composition of components handling cross-cutting concerns, with hooks used to weave and unweave advice at runtime proactively and retroactively. Moreover, as the WS components can be exposed with XML-based WSDL, the component composition can be specified with language neutral XML, which is further mapped to language-neutral type system CTS, with low-level CTS transparent to upper level composition decision makers. The experimental results show the profiling-based dynamic composition approach is encouraging with the appropriate control over the profiling scope in the WS scenario. Even though the approach presented in this paper is .NET based, the principle also applies to other platforms with adequate software vendor support.

With the different abstraction levels involved as shown in Figure 1, one future direction is to investigate the model-driven approach ([Cao-b, 05], [Frankel , 03], [Lédeczi, 01]) for modeling component composition concerns, so that component composition can be represented in high-level models which reduces the gaps between business requirements and underlying implementation, with AAR and rules as shown in Figure 3 automatically synthesized from models. We would also like to explore the use of mobile agents in the peer-to-peer component composition scenario where composition decisions can be federated and communicated seamlessly, for which security is also of vital concern in the future research.

Acknowledgements

This research is supported in part by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

References

[Brown, 00] A. W. Brown, Large-Scale Component-Based Development, Prentice Hall, 2000.

[Cao, 02] F. Cao, B. Bryant, R. Raje, M. Auguston, A. Olson, C. Burt, Component Specification and Wrapper/Glue Code Generation with Two-Level Grammar Using Domain Specific Knowledge, In Proc. Int. Conf. on Formal Engineering Methods, October 2002, 103-107.

[Cao-a, 05] F. Cao, B. R. Bryant, S.-H. Liu, W. Zhao, A Non-Invasive Approach to Dynamic Web Service Provisioning, In Proc. IEEE Int. Conf. on Web Services, July 2005, (*to appear*).

[Cao-b, 05] F. Cao, B. R. Bryant, W. Zhao, C. C. Burt, R. R. Raje, A. M. Olson, M. Auguston. Model-Driven Reengineering Legacy Software Systems to Web Services, 2005 (*submitted*).

[Chen, 01] Q. Chen, M. Hsu, Inter-Enterprise Collaborative Business Process Management, In Proc. Int. Conf. on Data Engineering, April 2001, 253-260.

[Clarke, 01] S. Clarke, R. J. Walker, Composition Patterns: An Approach to Designing Reusable Aspects, In Proc. Int. Conf. on Software Engineering, IEEE Computer Society, May 2001, 5-14.

[Clarke, 02] S. Clarke, Extending Standard UML with Model Composition Semantics, Sci. Comput. Program, 44(1), 2002, 71-100.

[Colan, 04] M. Colan, Service-oriented architecture expands the vision of Web Services, 2004, http://www-106.ibm.com/developerworks/webservices/library/ws-soaintro.html.

[Dan, 02] A. Dan, A. R. Franck, A. Keller, R. King, H. Ludwig, Web Service Level Agreement (WSLA) Language Specification, 2002, http://dwdemos.alphaworks.ibm.com/wstk/common /wstkdoc/services/utilities/wslaauthoring/WebServiceLevelAgreementLanguage.html.

[Devanbu, 96] P. Devanbu, S. Karstu, W. Melo, W. Thomas, Analytical and Empirical Evaluation of Software Reuse Metrics, In Proc. Int. Conf. on Software Engineering, IEEE Computer Society, March 1996, 189-199.

[Duzan, 04] G. Duzan, J. P. Loyall, R. E. Schantz, R. Shapiro, J. A. Zinky, Building Adaptive Distributed Applications with Middleware and Aspects, In Proc. Int. Conf. on Aspect-Oriented Software Development, March 2004, 66-73.

[Frankel , 03] D. S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley, 2003.

[Friedman-Hill, 05] E. J. Friedman-Hill, Jess 7.0, The Rule Engine for the Java Platform, Sandia National Laboratories, 2005.

[Gough, 02] J. Gough, Compiling for the .NET Common Language Runtime (CLR), Prentice Hall PTR, 2002.

[Heineman, 01] G. T. Heineman, W. T. Councill, Component Based Software Engineering: Putting the Pieces Together, Addison-Wesley, 2001.

[Hissam, 03] S. A. Hissam, G. A. Moreno, J. A. Stafford, K. C. Wallnau, Enabling predictable assembly, Journal of Systems and Software, 65(3), 2003, 185-198.

[Hölzle, 93] U. Hölzle, Integrating Independently-Developed Components in Object-Oriented Languages, In Proc. European Conference on Object-Oriented Programming, July 1993, 36-56

[Keller, 98] R. K. Keller, R. Schauer, Design Components: Towards Software Composition at the Design Level, In Proc. Int. Conf. on Software Engineering, April 1998, 302-311.

[Kiczales, 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, In Proc. European Conference on Object-Oriented Programming, June 1997, 220-242.

[Kiczales, 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An Overview of AspectJ, In Proc. European Conference on Object-Oriented Programming, June 2001, 327-353.

[Lédeczi, 2001] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer, 34(11), 2001, 44-51.

[Mezini, 98] M. Mezini, K. J. Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development. In Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications, October 1998, 97-116.

[Newcomer, 02] E. Newcomer, Understanding Web Services, Addison Wesley, 2002.

[Olson, 05] A. M. Olson, R. R. Raje, B. R. Bryant, C. C. Burt, M. Auguston, UniFrame-a Unified Framework for Developing Service-Oriented, Component-Based, Distributed Software Systems, Service-Oriented Software System Engineering: Challenges and Practices, Idea Group, 2005, 68-87.

[Pulvermuller, 99] E. Pulvermuller, H. Klaeren, A. Speck, Aspects in Distributed Environments, In Proc. Generative Component-based Software Engineering, September 1999, 37-48.

[Raje, 02] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt, A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components, Concurrency and Computation: Practice and Experience, 14(12), 2002, 1009-1034.

[Sahai, 02] A. Sahai, V. Machiraju, M. Sayal, L. J. Jin, F. Casati, Automated SLA Monitoring for Web Services, 2002, http://www.hpl.hp.com/techreports/2002/HPL-2002-191.pdf

[Seiter, 99] L. M. Seiter, M. Mezini, K. J. Lieberherr, Dynamic Component Gluing, In Proc. Int. Symposium on Generative Programming and Component-Based Software Engineering, September 1999, 134-164

[Stutz, 03] D. Stutz, T. Neward, G. Shilling, Shared Source CLI – Essentials, O'Reilly Press, 2003.

[Szyperski, 02] C. Szyperski, D. Gruntz, S. Murer, Component Software: Beyond Object-Oriented Programming, 2nd ed., Addison-Wesley/ACM, 2002.

[Yang, 02] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, P. K. McKinley, An Aspect-Oriented Approach to Dynamic Adaptation, In Proc. The First Workshop on Self-healing Systems, November, 2002, 85-92.

[Zhang, 03] C. Zhang, H.-A. Jacobsen, Refactoring Middleware with Aspects, IEEE Trans. Parallel Distrib. Syst. 14(11), 2003, 1058-1073.

GridFrame – A Framework for Building Quality Aware Component-based Grid Systems

Pradeep J. Mysore, Rajeev R. Raje, Purushotham V. Bangalore[1] and Barrett R. Bryant[1]

Department of Computer and Information Science,
Indiana, University Purdue University Indianapolis,
723 W. Michigan, St, SL 280H, Indianapolis, IN 46202,
USA.
pmysore@cs.iupui.edu, rraje@cs.iupui.edu

[1]Department of Computer and Information Sciences,
University of Alabama at Birmingham,
Birmingham, AL 35294
U.S.A.
puri@cis.uab.edu, bryant@cis.uab.edu

Abstract–**Predominantly, the Grid world has focused on discovering and using hardware solutions for executing scientific and mainstream applications. The applications for Grid are typically handcrafted and also assume the presence of an expert user, thus, making this process error-prone. This paper presents a framework, called GridFrame, whose vision is to reduce the complexity of the applications for Grid systems. GridFrame achieves this goal by providing an approach for semi-automatically discovering independently developed components and constructing quality-aware Grid applications using these components.**

I. Introduction

Software Component Frameworks [1] have been established as a standardized way of building commercial distributed applications from independently developed sub-units. However, the Grid world, which is predominantly scientific, has been slow in embracing these concepts [1, 2]. With increasing mainstream usage of Grid Computing, software component composition and reuse through service oriented Grids have become an increasing need for current and future Grid projects.

Despite the popularity of service oriented Grid, several interesting challenges, such as creating applications from pre-existing components, masking their heterogeneity, and reducing the manual involvement in the development phase, have yet to be tackled adequately. The current software development process for Grid applications consists of an initial application development, testing, and validation that is done on local resources with a subset of the program. After the initial validation, these applications are migrated to larger systems. All the necessary pieces for integration are hand-crafted and weaved manually to achieve a software realization of any application, thus, achieving little reuse. Also, there is no mature application development environment for the Grid - ad hoc approaches that are prevalent in the high-performance computing domain are used to develop Grid applications. Although, there are a few tools such as graphical modelers [3] that are available for aiding this process, it still requires a significant amount of manual intervention.

This paper describes a framework, GridFrame, for the creation and composition of distributed Grid components. Using GridFrame, programmers can reason about quality of service (QoS) for individual Grid services as well as a constructed Distributed Computing System (DCS) out of these Grid services. The ability to compose and deploy grid-enabled applications from pre-existing components will enable the rapid design and development of next generation distributed applications while promoting better software reuse with the creation of domain-specific component repositories.

II. Related Approaches

A *Grid experience* is defined as the Grid utilization process, which runs from the Grid application creation to the final deployment and execution of the application. Most of the existing approaches, such as [4, 5] are targeted at the latter, i.e., deployment and execution of the application, and tackle challenges such as requirements analysis, selecting hardware resources and providing middleware facilities for enabling a user to deploy a Grid application(s). Typically, these approaches assume that the Grid application has already been designed and pre-customized to the Grid deployment phase. Only a few approaches (e.g., [3, 6, 7, and 8]) address the challenges of providing a software component framework for creating component-based Grid applications using pre-built components.

Before elaborating on any of the current Grid software component framework approaches, it is necessary to first identify the requirements and constraints that the Grid places on such a framework. In the current Grid scenario (and for purposes of this paper), components are defined to be Open Grid Services Architecture (OGSA) [2] Services deployed in OGSA containers with associated service data indicating their characteristics. These components are characterized by their dynamic nature, implying that they might be available for varying intervals of time, with frequent changes of their availability status. Also, the components tend to be heterogeneous and are distributed in nature. Hence, a software component framework [1] for Grid must fulfil the following requirements; a) Should be able to tackle heterogeneity in language, model, technology and architecture, etc., b) Allow a way of dynamic discovery of components, c) Provide a means for ascertaining non-functional attributes such as QoS of individual components as well as the integrated system, and d) Provide a user friendly mechanism for the system integration.

In the current Grid scenario, a user developing a Grid application from pre-built components has to either write scripts in a XML representation [6], write scripts in a domain specific language [7, 9], or employ application workflow diagrams and graphical modelers [3, 8].

Conforming to the first approach, ICENI [6] provides a component based framework for creating Grid applications from pre-built components, discovered from private as well as public meta-repositories. Whenever a new component is developed, a component specification is created in terms of a CXML (Component – eXtensible Markup Language) [10] document, describing the component's behaviour and interface. Implementations of the specification are placed in meta-repositories, with meta-data describing their performance characteristics and resource requirements. Based on a problem definition, composition of these implementations to form a Grid application is described in terms of an application description document, which is a CXML specification of the complete component composition. At runtime, the application description document is converted into an active Java representation by utilizing the component specification meta-data within the repository. The run-time representation is used to map the application requirements into available resources, based on requirements' and implementations' meta-data. While this approach does attempt to provide a component based Grid framework, for satisfying a few of the aforementioned requirements for a Grid framework, it does not succeed on several fronts. Firstly, it does not tackle heterogeneity at the component model level, only at the language level. The component CXML document does not provide a comprehensive enough QoS catalogue for comparing and matching components attributes, or for prediction at the component and system level. Since CXML does not accord the flexibility to express a application in terms of a hierarchy of possible subsystems, even a small change in the problem definition implies that the application CXML has to rewritten.

One script based approach incorporated in GRADS [9] aims at providing domain specific high-level programming systems for problem solving environments, by which end users can rapidly develop new applications using standard notations of their problem domains. Here, the pre-built components are organized into optimized libraries, using a set of library design and specification strategies. Also, the application library is annotated with the following details; a) program transformation specifications detailing how program sequences can be replaced with equivalent, but more efficient sequences and b) sample calling programs illustrating typical usage patterns. In a separate step, mappings from scripting languages to library implementation language are provided. The scripting languages enable the usage of components as primitive objects and define operations on them. A translator generator processes the enhanced library for hours or days and produces an executable. Using any of the allowed scripting languages, the user has to write an application script involving operations, initiation and configurations of the primitive objects to construct a Grid application. The scripts are then translated and compiled using the above domain specific translator. While the approach promises significant improvement in performance issues, it does not provide a software component framework (as in [1]) per se. As a result, there are no facilities for discovery of components, prediction of QoS of individual components and integrated system. Even though end users can develop new applications using their domain specific notations, there is a significant amount of latency curve associated with learning a new scripting language. The developed applications are invariably individualistic handcrafted solutions.

XCAT3 [7] is another script based framework that emphasizes distributed computing and provides Grid and Web Services connectivity to CCA (Common Component Architecture) [11] based on the OGSA. It provides a component based framework by which components (CCA components and/or Grid services based on OGSA) can be instantiated and connected together. Each component contains *provides-ports* indicating the functionality the component provides to other components and *uses-ports* indicating the needed functionality from other components that the component needs to function. As a result, each component in the XCAT3 framework consists of port interfaces, port implementations and an implementation. *Builder* services APIs in Java are provided, by which instances of components can be created and composed together to form a distributed application. Also, APIs for querying services of components, destroying component instances, and invoking methods on instantiated components are provided. While this approach is promising, the resulting applications are again handcrafted solutions. Further details of XCAT3 and OGSA in particular, are presented in the Section IV.

In an application workflow approach, for example in CrossGrid [8], the user supplies an initial application workflow document, detailing the components, their interactions and the workflow. Here, components are CCA-based, are developed independently, and are registered with OGSA registries. A flow composer parses the user workflow diagram, performs component lookups based on Port type or ID attributes and builds different final workflow documents with every distinct set of matched components. Finally, the user can choose a final workflow document corresponding to his view of the integrated system. This approach has intrinsic limitations similar to the script based approaches in that it places undue importance on the expertise of the user, does not offer any QoS testing and finally results in handcrafted, individualistic solutions with restricted reusability. In Triana [3], graphical modelers and toolkits are employed, which provide the user a higher level of abstraction than application workflows. A user creates an application by dragging, dropping and deleting components and associated relationships in a graphical window. Here, though a greater level of convenience than the application workflow approach is accorded to the user, the other workflow limitations still remain. Other approaches like ECSF [12] provide a distributed computing paradigm suitable for multidisciplinary Grid applications, but are also limited by the same problems since

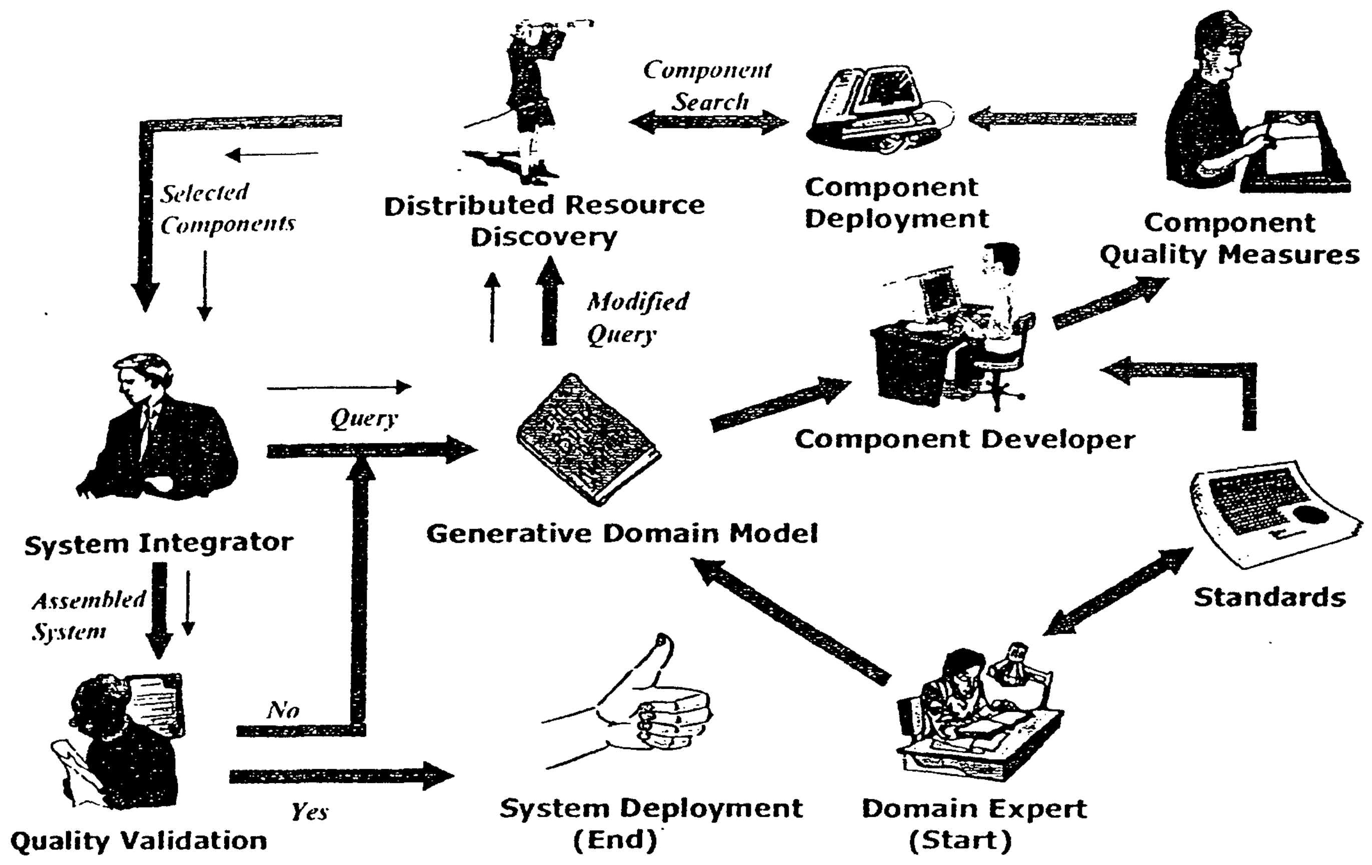


Fig. 1. GridFrame Process

their underlying principles are not based on the component paradigm. GridLab [13] does not provide a component creation framework, but focuses on providing high level application toolkits that interface between user applications and Grid middleware packages like Globus [2].

To summarize the related approaches, many challenges such as heterogeneity of components, their resource discovery and QoS prediction etc., associated with creating a component based Grid framework have currently not been addressed satisfactorily. If Grid has to become omni-present, in both scientific and commercial domains, these challenges need to be effectively addressed. One possible approach to addressing these challenges is by creating a comprehensive framework, incorporating solutions to the indicated problems.

III. GRIDFRAME APPROACH

As a part of a related effort called UniFrame [14], the principles behind addressing some of the aforementioned challenges have been developed. UniFrame is a component-based framework for interoperation of heterogeneous distributed components. In this paper, a symbiosis of the principles of UniFrame and Grid to form a component based Grid framework known as *GridFrame* is proposed. The key research issue that GridFrame addresses is the conception of a semi-automated Component Based Grid System (CBGS) development process involving the dynamic discovery of distributed Grid components, generation of the composed system and validation of quality requirements

GridFrame differs significantly from current Grid approaches by relying on an expert created generative domain model (GDM) [15]. Experts from the particular domain create the GDM containing the details of the distributed Grid application under consideration. The GDM contains details of the software architecture of families of possible systems in terms of the constituent software components, descriptions of the component characteristics and interactions, rules for the prediction and monitoring of quality of the constituent components as well as the integrated system. A reliance on a GDM has many advantages; a) it is created by domain experts, thus, end users are abstracted from domain knowledge expertise and required skills, b) model for component developers to create individual components, and c) it provides rules for the composition and decomposition of components with associated quality of service. A component developer for an application consults the GDM and creates component implementations using the listed specifications. It is anticipated that many such components for a particular application with possibly different QoS, will be developed and deployed over a network.

A domain expert creating a GDM for a Grid application has to follow the GDM development process as outlined in [16]. The GDM development process consists of three phases; i) domain analysis - establishing domain scope, identification of functional and QoS requirements and mapping of relevant domain concepts, ii) domain design - development of common layered architecture for a family of possible systems and QoS related models, and iii) ordering design - design of ordering schemes for ordering a component-based system from the family of possible systems. Using this process, a GDM for the domain is developed. The GDM consists of three parts: general information, which includes a description for the modelled domain; a problem space, used by an application programmer to specify the needs; and a solution space, which contains various models including configuration knowledge to provide solutions for a CBGS family. Further details of the GDM are given in the case study section.

Component specifications are described by an associated Unified Meta-component Model (UMM) [17]. UMM has three parts: a) components, b) service and its guarantees, and c) infrastructure. A component in UMM is considered to be a tuple consisting of: a) inherent attributes - bookkeeping information such as name, description, etc., b) functional attributes - interface, pre-post conditions, algorithms, etc., c) non-functional attributes - supported QoS parameters and values with corresponding deployment environments, d) cooperative attributes - details of the collaborations of systems in which the component participates, e) auxiliary attributes - special features such as mobility, security, fault-tolerance, etc., and f) deployment attributes - configuration, initialization information.

The second part of the UMM is the service and associated guarantee of delivering that service. While realizing a CBGS from a set of independently created components, it is necessary to reason about the quality of the integrated CBGS. The quality of the integrated system translates into the quality of service offered by each component and of their interactions. Hence, it is necessary that a component provide a pre-determined level of quality of both its functional and non-functional features. For doing so, the UMM requires a component developer to specify the QoS parameters that are applicable to a particular component and the ranges that the component can guarantee when operating under a certain execution environment [17].

The third part of the UMM is the infrastructure that supports the creation, publication, deployment, and location of the components and their services. This infrastructure is provided by Grid Resource Discovery System (GRDS) based on [18], which is the infrastructure that supports the creation, publication, deployment, and location of the components and their services. The discovery process in GRDS is scoped administratively implying that it locates services within an administratively defined logical domain. A domain is defined as industry specific markets such as Information Filtering Services, Health Care Services, and Financial Services, etc. The GRDS architecture consists of the following entities: a) head-hunters for discovering component specifications, b) containers for component registration, and c) components. Components are implemented in accordance with component models such as Microsoft .NET, Java RMI, CORBA, etc., and are registered with the binding service of that model. The binding services are modified Grid containers such as Globus J2EE container, .NET container etc. Headhunters periodically communicate with these Grid containers and retrieve and store specifications (service data) of registered components into their local meta repositories. For more details about UMM components, service and infrastructure, please refer to [14], [19] and [18] respectively.

Components offer services, indicate and guarantee the quality of their services, and hence, it is necessary to facilitate the publication, selection, measurement and validation of the component and system QoS values. The Grid Quality of Service Framework (GQoS) based on [17] provides the necessary guidelines for the component developers and system developer using GridFrame. The GQoS is made up of three parts: a) QoS catalogue – collection of possible QoS parameters such as end-to-end delay, throughput etc., b) specification and measurement of QoS and c) composition/decomposition models for QoS parameters. For further details, please refer to [14] and [18].

Fig. 1 gives an illustration of the GridFrame process. In the beginning of the process, a Grid system developer, developing a CBGS, for a specific application issues a query containing the requirements for the CBGS. The query can comprise of functional requirements as well as non-functional QoS requirements such as end-to-end delay, throughput, etc. The query processing consults the GDM for the design of an appropriate CBGS and may divide the query into many sub-queries, each corresponding to a single component UMM specification. These sub-queries are passed to the GRDS which searches for appropriate matching components. If components are found, they are displayed to the system developer. The system developer decides on the components to be included in building the system, based on various criteria such as offered QoS. Also, each component provides an associated testing mechanism, which can be used to dynamically test the QoS characteristics of the component. These dynamic test values can be judged against the component developer's specifications of the component. After the system developer selects his choice of components, the generation of the integrated system is carried out by the GridFrame System Integrator [16] using the selected components.

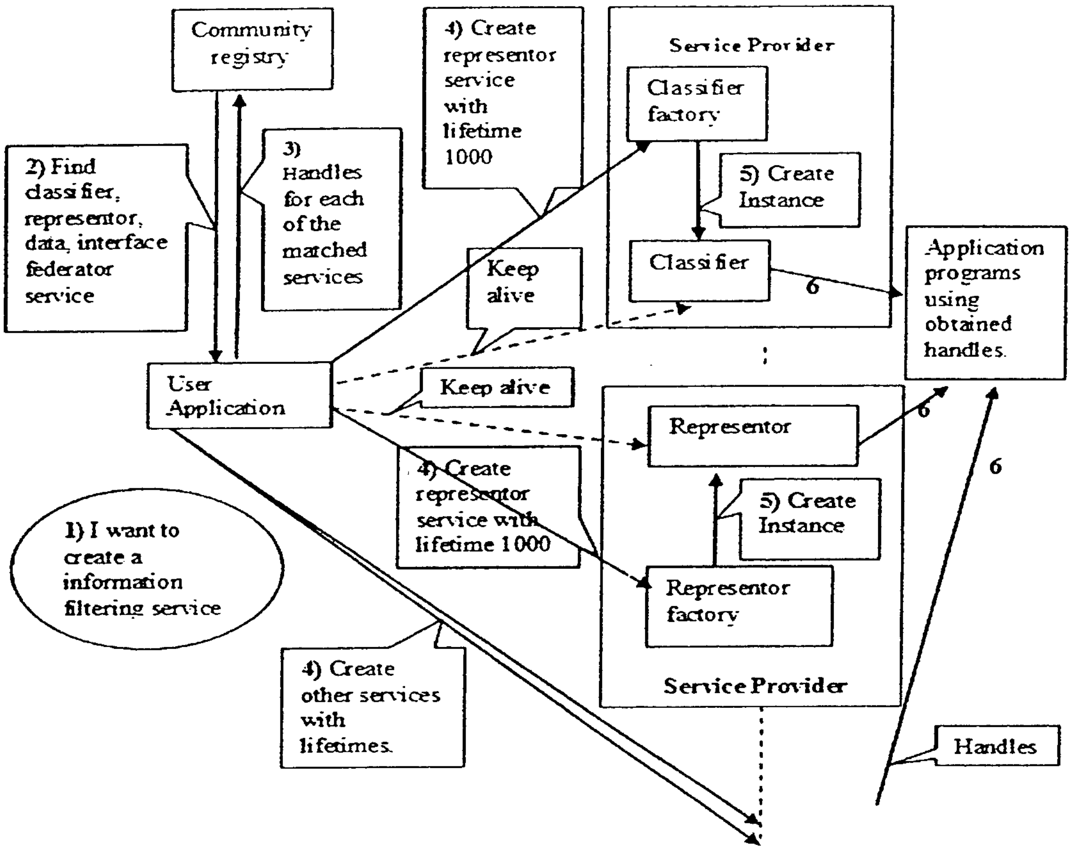

Community registry
4) Create representor service with lifetime 1000
Service Provider
Classifier factory
5) Create Instance
Classifier
2) Find classifier, representor, data, interface federator service
3) Handles for each of the matched services
Keep alive
6
Application programs using obtained handles.
User Application
Keep alive
Representor
6
4) Create representor service with lifetime 1000
5) Create Instance
1) I want to create a information filtering service
Representor factory
Service Provider
4) Create other services with lifetimes.
Handles

Fig. 2. Creating DIFS using OGSA Grid Services11/28/04

The system developer uses the principles of two-level grammar and event grammars [14] to generate the necessary glue for the creation of the integrated system. The composition models present in the GDM can be used by the system developer to predict the quality features of the integrated system. Also, the instrumentation code present in the glue, that is created based on event grammars, allows a dynamic measurement of quality features of the integrated system. These dynamic values are compared against the static predictions and if there is a match, the assembled Grid system is deployed and is ready to use.

IV. Case Study

Out of approaches described in Section II, using OGSA services to leverage existing services to form complex distributed solutions is the popular option now. To contrast the OGSA based approach with the GridFrame approach, a case study from the domain of distributed information filtering is considered. Typically, a distributed information filtering service (DIFS) reduces information overload by supporting personalization of long term information needs of a particular user or group of users with similar needs. Here, a DIFS based on DSIFTER [20] is considered, in which one of the authors was involved. Using user profiles and periodic feedback, the DIFS rank-orders documents and performs a mapping from the space of documents to the space of user relevance values. Typically, in a DIFS, more often than not, documents exist at diverse sites and are received by the user through disparate, independent channels. The task of storing such documents, before filtering is handled by a data acquisition service (DAS). A representation service (RS) converts these stored documents into structures, which can be efficiently parsed without the loss of vital content. A classifying service (CS) classifies these stored structures using clustering algorithms on the basis of user interests specified in a user profile service (UPS). The UPS is continuously updated using reinforcement learning algorithms to reflect current user interests. A user interface service (UIS) displays the ranked documents and collects user feedback for user profile learning. A federation service (FS) enables interconnection of DIFS systems.

A. *Creating a DIFS system using OGSA Grid Services*

1. Assuming a complete DIFS service is not available, a Grid user has to decompose his requirements to form a list of the previously identified Grid services that would aggregate to form a DIFS Grid service. Fig. 2 illustrates an example of the process by which a user can build a DIFS system using Grid services. It contains the following steps:

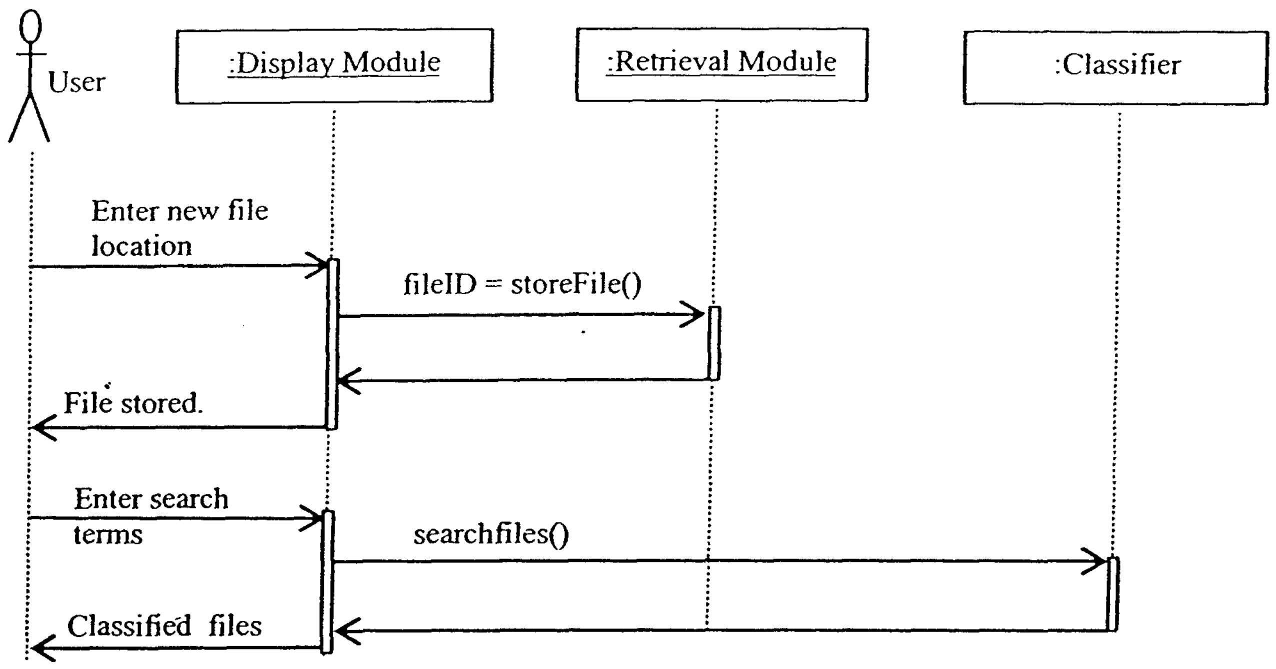


Fig. 3. Partial Sequence Diagram for Search

2. The user contacts a known registry to identify service providers who can provide the required services and presents a list of requirements including cost and performance.
3. The handles for needed service factories that match user requirements are returned to the user.
4. The user supplies instantiation details such as needed operations, etc., and initial lifetimes for the service instances.
5. If agreeable, the service providers create service instances with user supplied details.
6. Using the service handles, the user writes application programs for aggregating the services to form a DIFS system.

Visions of enterprises using Grid Services approach to dynamically compose new applications such as above to address the specific needs of the business at any point in time have been painted. But there are several limitations with this approach, particularly in regard enterprise applications. The resulting new applications are basically handcrafted solutions with limited reuse. Any slight change in the problem definition, for instance using a .NET display component, if a previously used Java component is not available, will entail a complete rewrite of the previous solution. Also, users do not have options for any preliminary testing of the integrated application, implying that they cannot make any informed decisions about the QoS of the integrated application before the actual deployment. In addition, most of the techniques for discovery of components assume that the components are homogeneous in nature and rely on simple interface matching and component context dependencies, which is not sufficient enough for a process, which is a precursor for composing high confidence Grid systems. Also, the approach assumes a high level of programming skill of users, which is typically not the case with mainstream Grid users. These are serious drawbacks, particularly considering that mainstream domains such as enterprise applications have stringent requirements about quality and reusability of applications.

B. Creating a DIFS system with GridFrame

For the sake of brevity, the focus is mainly on the overall outline of carrying out the development of a partial GDM as well as the discovery mechanisms of GridFrame possibly resulting in the omission of in depth details, which can be referred to, using the associated references.

1) GDM process: Due to space constraints, only some of the aspects of GDM like feature diagrams and use-cases depicting the configuration knowledge of the integrated system are shown here. In the GDM, the components making up the DIFS system are identified along with their functional characteristics such as required interfaces, provided interfaces, etc., and non-f'nctional characteristics such as QoS metrics. In addition to the feature diagram, the GDM contains sequence diagrams, which capture the behavioural aspects of the system. Sequence diagrams, such as Fig. 3 illustrate the interaction of components in the system with each other as well as with users. Fig. 5 shows a feature diagram illustrating the DIFS family of sub-systems which can be possibly built with the identified components.

The details about the concept of features and the notation used for describing a feature diagram are proposed in [16]. The given feature diagram indicates possible architectural alternatives for a DIFS. For example, two possible alternatives for a DIFS could be: version (a) made up of RM, TM, RP, CL, SP, TI and version (b) made up of WM, RM, TM, RP, CG, CL, CP, DM, SM, EM, GM, ECM, CM. As indicated earlier, depending upon the input query presented by the system integrator, an appropriate alternative will be selected during the system development process. Each node in the feature diagram indicates an abstract component, which will be described by its corresponding UMM specification. The component specification in UMM is a multi-level contract [19] with bookkeeping information such as component id, domain name, and algorithmic, technological information such as function name, algorithm name etc. For example, a partial UMM-specification for a typical classifier could be:

1. Component Name: Classifier
2. Domain Name: Information Filtration
3. System Name: InformationFilter
4. Informal Description: Provide classification service for documents.
5. Computational Attributes:

5.1 Inherent Attributes:

5.1.1 id: N/A 5.1.2 Version: version 1.0 5.1.3 Author: N/A

5.1.4 Date: N/A 5.1.5 Validity: N/A 5.1.6 Atomicity: Yes

5.1.7 Registration: N/A 5.1.8 Model: N/A

5.2 Functional Attributes:

5.2.1 Function description: Act as classification server for documents in system.

5.2.2 Algorithm: N/A 5.2.3 Complexity: N/A

5.2.4 Syntactic Contract

5.2.4.1 Provided Interface: IClassification

5.2.4.2 Required Interface: NONE

5.2.5 Technology: N/A

5.2.6 Expected Resources: N/A

5.2.7 Design Patterns: NONE

5.2.8 Known Usage: Classification of documents

5.2.9 Alias: NONE

6. Cooperation Attributes:

6.1 Preprocessing Collaborators: Representor

6.2 Postprocessing Collaborators: NONE

7. Auxiliary Attributes:

7.1 Mobility: No 7.2 Security: L0 7.3 Fault tolerance: L0

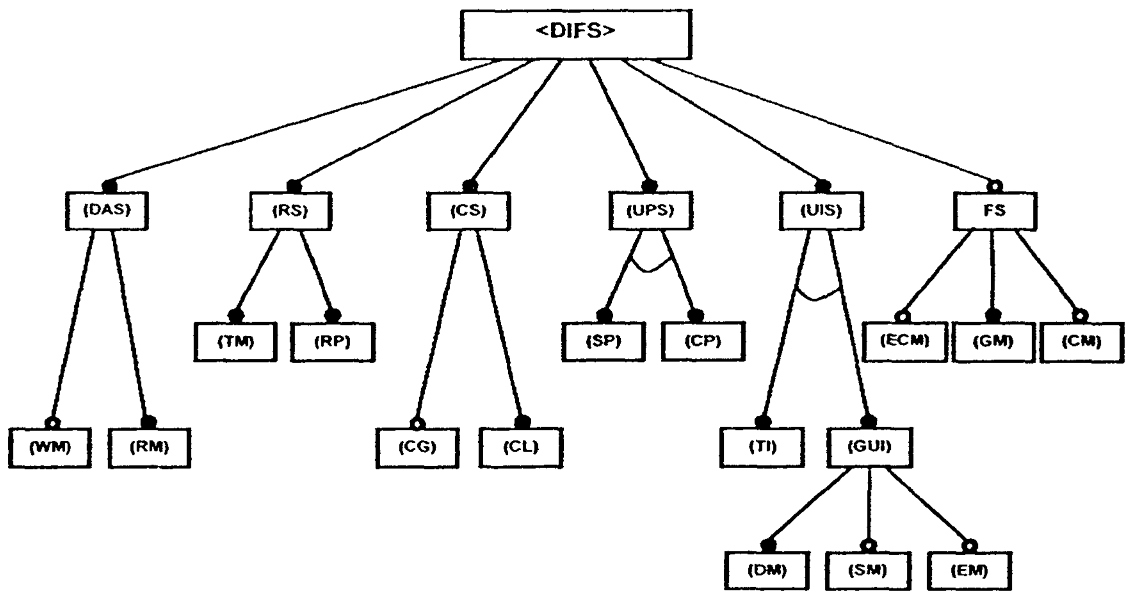
8. Quality of Service Attributes

8.1 QoS Metrics: throughput, end-to-end delay

8.2 QoS Level: N/A 8.3 Cost: N/A 8.4 Quality Level: N/A

8.5 Effect of Environment: N/A 8.6 Effect of Usage Pattern: N/A

9. Deployment Attributes: N/A



Legend:

DAS – Data Acquisition Service	**SP – Simple Profiler**	**CL – Classifier**
WM – Wrapper Module	**CP – Complex Profiler**	**UPS – User Profile System**
RM – Retrieval Module	**UIS – User Interface Service**	**GM – Group Manager**
RS – Representer Service	**TI – Text Interface**	**EM – Editing Module**
TM – Thesaurus Module	**GUI – Graphical Interface**	**FS – Federation System**
CS – Classifying Service	**DM – Display Module**	**ECM – Economic Module**
CG – Centroid Generator	**SM – Statistical Module**	**CM – Common Module**

Fig. 5. Feature Diagram of DIFS

Once the GDM has been developed, component developers are free to develop and deploy components using their choices of technology, language, etc., according to the specifications in the GDM. The developed concrete components have to strictly adhere to the GDM abstract specifications, but can be implemented in different technologies, algorithms etc. with corresponding QoS values. For example, one Representor Module (RM) can be implemented in .NET technology using a vector space model [20] with 340 ms turnaround time while another RM can be implemented in Java RMI using a different model with 320 ms turnaround time, with corresponding QoS attributes.

2) Discovery of components and Integration of the system: After the creation of the GDM and deployment of components, a system developer can query for an instance of a system using a tabular graphical interface [16], containing different options for the different possible systems. For example, the options could be a basic DIFS with minimal functionality incorporating instances of RM, TM, RP, CL, SP, TI and GM or an advanced DIFS with increased functionality incorporating instances of WM, RM, TM, RP, CG, CL, CP, DM, SM, ECM, GM. For example, the system developer might query for a simple DIFS with QoS values such as the maximum permissible end-to-end delay and minimum throughput for the system specified as 1800 ms and 400 op/s respectively. Using the decomposition model in [22], the given QoS requirements for the whole system are decomposed into the QoS requirements for each of the constituent components. By means of the GDM and the QoS requirements, for each of the components making up the chosen system, a query is created. These queries are presented to GRDS for discovering concrete instances of the components, which can match the requirements.

When the GRDS receives the requests, a subset of headhunters in the specified domain (in this case, distributed information filtering) is contacted for concrete instances of the components. These headhunters search their local meta-repositories and perform syntax, semantic and QoS matching of the stored specifications with the queries. Each query has an associated timestamp, depending on which the queries can be propagated to other headhunters. For details about selection, propagation and matching algorithms of headhunters, see [18]. As explained in the GridFrame process, the system developer chooses from among the listed components on basis of QoS values, (available from the service data), and uses the GridFrame System Integrator to test and build the integrated system.

A brief comparison of the two approaches suggests the following:

1. As opposed to handcrafting, the use of a GDM in GridFrame enables the creation of standardized solutions by which the reusability of individual components as well as the integrated system is improved.
2. Quality of service theme is maintained throughout the GridFrame process, as a result of which predicting and monitoring of component performance at the component level as well as system level is possible.
3. GridFrame accommodates heterogeneity by which components can be implemented in different models and technologies.
4. By providing a semi-automated framework for composing services, GridFrame ensures that user intervention is minimized, enabling novice end users to integrate systems.

V. Conclusion

The proposed framework provides a semi automated approach for building Grid systems from pre-built Grid services using concepts of software engineering. Using the framework, it is possible for end users to both predict and reason about the quality of the integrated system as well as the individual services. Although a simple example is provided here, the principles are general enough to be applicable for both mainstream and research Grid projects. The development of Grid systems involves both construction and deployment of the system. Here, only the construction issues of a component based Grid system using a GDM were discussed. Utilizing the GDM for deployment issues such as assessing hardware resource requirements, selecting ideal resources, etc., is the focus for current research. The GridFrame process has been investigated using the Globus toolkit and the current UniFrame infrastructure by means of trivial examples. While the results are promising and show that such a process is plausible, validation on a realistic, large scale scientific or mainstream Grid application is one of the key research goals for current and future efforts.

References

[1] C. Szyperski, D. Gruntz,, S. Murer, *Component Software - Beyond Object-Oriented Programming*, Second Edition. Boston: Addison-Wesley/ACM Press, 2002.

[2] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Open Grid Service Infrastructure WG, *Global Grid Forum*, June 22, 2002.

[3] I. Taylor, M. Shields, I. Wang, and R. Philp, "Distributed P2P computing within triana: A galaxy visualization test case," *IPDPS*, 2003.

[4] G. Allen, W. Benger, T. Goodale, H. Hege, G. Lanfermann et al, "The Cactus Code – A Problem Solving Environment for the Grid," *Proceedings of the 9 th IEEE Int'l. Symposium on High Performance Distributed Computing*, Pittsburgh. 2000.

[5] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid", *Grid Computing: Making The Global Infrastructure a Reality*, John Wiley, 2003.

[6] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field and J. Darlington, "An Integrated Grid Environment for Component Applications," *Second International Workshop on Grid Computing 2001*, pp. 26-37, November 2001.

[7] [7] S. Krishnan, and D. Gannon, "XCAT3: A Framework for CCA Components as OGSA Services," *Proceedings of HIPS 2004*, April 2004.

[8] M. Bubak, K. Gorka, T. Gubala, M. Malawski, K. Zajac, "Automatic Flow Building for Component Grid Applications," *Fifth International Conference on Parallel Processing and Applied Mathematics*, 2003, in press.

[9] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon et al, "Telescoping Languages: A Strategy for Automatic Generation of Scienti c Problem-Solving Systems from Annotated Libraries," *JPDC* , Vol. 61, No. 12, pp. 1803-1826, Dec 1, 2002.

[10] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field and J. Darlington, "Optimisation of Component-based Applications within a Grid Environment," SuperComputing 2001, Nov 2001.

[11] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, and A. Slominski, "On Building Parallel and Grid Applications: Component Technology and Distributed Services," *CLADE*, 2004. [12] P. Bangalore, "An Open Framework For Developing Distributed Computing Environments For Multidisciplinary Computational Simulations," *PhD thesis*, Mississippi State University, May 2003.

[12] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser et al, "The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid," Unpublished.

[13] R. Raje, M. Auguston, B. Bryant, A. Olson, C. Burt, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," *Proceedings of the 2001 Monterey Workshop*, pp. 109-119, Monterey, California, 2001.

[14] K. Czarnecki. and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[15] Z. Huang, The UniFrame System-level Generative Programming Framework. *MS thesis*, IUPUI, CIS Department, 2003.

[16] G Brahnmath, R. Raje, A. Olson, B. Bryant, M. Auguston, C. Burt,

[17] "A Quality of Service Catalog for Software Components," *Proceedings of the Southeastern Software Engineering Conference*, Huntsville, Alabama, 2002.

[18] N. Siram, "An Architecture for the UniFrame Resource Discovery Service," *MS thesis*, IUPUI, CIS Department, 2002.

[19] A. Beugnard., J. Jezequel, N. Plouzeau. and D. Watkins, Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.

[20] R. Raje, M. Qiao, S. Mukhopadhyay, M. Palakal, J. Mostafa, "SIFTER-II: A Heterogeneous Agent Society for Information Filtering," *Proceedings of ACM Symposium on Applied Computing*, SAC'01, pp: 121-123, Las Vegas, Nevada, 2001.

[21] C. Sun, "QoS Composition and Decomposition Model in UniFrame," *MS thesis*, IUPUI, CIS Department, 2003.

Service-Oriented Software System Engineering

Challenges and Practices

Zoran Stojanovic & Ajantha Dahanayake

Chapter IV

UniFrame: A Unified Framework for Developing Service-Oriented, Component-Based Distributed Software Systems

Andrew M. Olson
Indiana University Purdue University, USA

Rajeev R. Raje
Indiana University Purdue University, USA

Barrett R. Bryant
University of Alabama at Birmingham, USA

Carol C. Burt
University of Alabama at Birmingham, USA

Mikhail Auguston
Naval Postgraduate School, USA

Abstract

This chapter introduces the UniFrame approach to creating high quality computing systems from heterogeneous components distributed over a network. It describes how this approach employs a unifying framework for specifying such systems to unite the concepts of service-oriented architectures, a component-based software engineering methodology and a mechanism for automatically finding components on a network in order to assemble a specified system. UniFrame employs a formal specification language to define the components and serve as a basis for generating glue/wrapper code that connects heterogeneous components. It also provides a high level language for the *system developer to use for inserting code in a created system to validate it empirically and estimate the quality of service it supports. The chapter demonstrates how a comprehensive approach, which involves the practicing community as well as technical experts, can lead to solutions of many of the difficulties inherent in constructing distributed computing systems.*

Introduction

The architecture of a computing system family can be represented by a business model comprising a set of standard, platform independent models residing in a service layer, each of which is related to a platform specific model that corresponds to one or more specific realizations of the service. A system is realized by assembling the realizations according to the specified architecture. This Service-Oriented Architecture offers many advantages, such as flexibility, in constructing and modifying a computing system. Because business requirements can change rapidly, both the services making up a business model and their platform specific realizations may need to change rapidly in response. With an agile mechanism to trace out an appropriate architecture, the development engineer can react quickly by building a modified realization of the system. Nevertheless, there are many practical issues that make effecting this process difficult. For example, an environment in which this approach has greatest appeal is typically distributed and heterogeneous. This makes the mapping of a system's platform independent model to a platform specific model (Object Management Group, 2002) quite complex and subject to variation.

This chapter describes the basic principles of the UniFrame Project, which defines a process, based on Service-Oriented Architecture, for rapidly constructing a distributed computing system that confronts many of these inherent difficulties. UniFrame's basic objective is to create a unified framework to facilitate the interoperation of heterogeneous distributed components as well as the construction of high quality computing systems based on them. UniFrame combines the principles of distributed, component-based computing, Model-Driven Architecture, service and quality of service guarantees, and generative techniques.

Though better than handcrafting distributed computing systems, developing them by composing existing components still poses many challenges. A comprehensive treatment of these and the corresponding solutions that UniFrame proposes exceeds the scope of this chapter, so it sketches the features of UniFrame that are most related to the book's service-oriented engineering theme along with references to further reading.

Background

Despite the achievements in software engineering, development of large-scale, decentralized systems still poses major issues. Recent experience has demonstrated that the principles of distributed, component-based engineering are effective in dealing with them. Weck (1997), Lumpe, Schneider, Nierstrasz, and Achermann (1997), and the works of Batory et al., for example, Batory and Geraci (1997), concern the composition of components. The approach of Griss (2001) to developing software product lines is similar to UniFrame's, except that UniFrame avoids descending to code-fragment-sized components. Brown (1999) surveys component-based system development, whereas Heineman and Councill (2001) and Szyperski, Gruntz, and Murer (2002) provide extensive discussions of different aspects.

Heineman and Councill (2001) provide a general definition of a component model. Many different models for distributed, component-based computing have been proposed and implemented. Among these, J2EE™ (Java 2 Enterprise Edition) and its associated distributed computing model (Java-RMI), CORBA® (Common Object Request Broker Architecture), and .NET® have achieved the greatest acceptance. Typically, each prevalent model assumes the presence of homogeneous environments; that is, components created using a particular model assume that any other components present adhere to the same model. For example, the white paper on Java Remote Method Invocation (2003) describes RMI as an extension of Java's basic model to achieve distributed computation, assuming, thus, an environment consisting of components developed using Java and communicating with each other using method calls. Schmidt (2003) provides an overview of CORBA, which indicates that CORBA does provide a limited independence from the components' development language and deployment platform by specifying components with an interface definition language. This permits implementation in any languages for which mappings with the interface definition language exist. Again, an implicit assumption is that, typically, a CORBA component will communicate with another CORBA component. Microsoft's .NET is intended as a programming model for building XML™-based Web services and associated applications. It provides language independence with an interface language and a common language runtime (Microsoft .NET Framework, 2003). The implicit assumption of homogeneity still holds.

UniFrame

Current approaches for tackling heterogeneity are *ad hoc* in nature, requiring handcrafted software bridges, so have many drawbacks. It is difficult to make components of different models interoperate, and handcrafting is known to be error prone. Moreover, dependence on a single model meshes poorly with the grand notion of a component (or services) bazaar over a distributed infrastructure, as the success of such a bazaar requires local autonomy for deciding various policies, including the choice of the underlying model. Thus, there is a need for a framework, such as UniFrame, that will support seamless interoperation of heterogeneous, distributed components. UniFrame consists of:

- the creation of a standards-based meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components;

- an automatic generation of glue and wrappers for achieving interoperability;
- guidelines for specifying and verifying the quality of individual components;
- a mechanism for automatically discovering appropriate components on a network;
- a methodology for developing distributed, component-based systems with service-oriented architectures; and
- mechanisms for evaluating the quality of the resulting component assemblages.

UniFrame creates more general distributed systems than the point-to-point interactions of current Web services and also emphasizes determining the Quality of Service (QoS) during system assembly. For pragmatic reasons, UniFrame provides an iterative, incremental process for assembling a distributed computing system (DCS) from services available on the network that permit selecting among alternative components during system construction. In order to increase the assurance of a DCS, UniFrame employs automation, to the extent feasible, in the processes of locating and assembling components, and of component and system integration testing. The ICSE 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction (Crnkovic, Schmidt, Stafford & Wallnau, 2003) focused on automated composition theories in constructing a DCS. Although automation is a goal of UniFrame, it presently focuses on the more practical, implementation aspects.

Unified Meta-Component Model (UMM)

Because future service-oriented systems will consist of independently developed components adhering to various models, a meta-model that abstracts the features of different models, enhances them and incorporates innovative concepts, is necessary in order to facilitate their creation. Raje (2000) and Raje, Auguston, Bryant, Olson, and Burt (2001) describe a central concept of UniFrame, the Unified Meta-component Model, that does this. It consists of three parts: (a) components, (b) service and its guarantees, and (c) infrastructure. These are not novel separately, but their structure, integration, and interactions form the UMM's distinguishing features. Components in the UMM have public interfaces and private implementations, which may be heterogeneous. Each interface comprises multiple levels. In addition to emphasizing a component's functional responsibilities (or the services it offers), the UMM requires component developers to advertise and guarantee a QoS rating for each component. The UMM's infrastructure supplies the environment necessary for developing, deploying, publishing, locating, assembling, and validating individual components and systems of components. The following subsections expand upon these concepts.

Component

The UMM defines a component as a sextuple consisting of the attributes (inherent, functional, nonfunctional, cooperative, auxiliary, deployment). This view of a component conforms to the definition of Szyperski, Gruntz, and Murer (2002). The inherent attributes contain the bookkeeping information about a component, such as the author, the version, and its validity period. The functional attributes of a component contain its interface, along with the necessary pre- and post-conditions, and component model of any associated implementation. They also indicate related details, such as algorithms used, underlying design patterns and technology, and known usages. The nonfunctional attributes represent the QoS parameters supported by the component, along with their values that the component developer guarantees in a specific deployment environment. These attributes may also indicate the effects of the deployment environment and usage patterns on the QoS values. The cooperative attributes describe how components actively collaborate, exchanging services. The auxiliary attributes exhibit other characteristics, such as mobility, various security features, and fault tolerance that the components may possess. A component needs deployment rules, specified in the deployment attributes so that it can be configured, initialized, and made available on a network.

Service

As described by Raje (2000), this part of the UMM consists of the computational tasks and guarantees that a component performs. To realize a DCS from a set of independently created components, the system integrator needs to reason from the service assurance of each component to obtain the assurance of the integrated DCS. Hence, a component must provide a predetermined level of assurance of both its functional and nonfunctional features. Various techniques, such as formal verification, have been proposed for reasoning about the functional assurance of a DCS. Therefore, the UMM assumes the use of an appropriate mechanism for functional assurance. The UniFrame research focuses on assuring the nonfunctional features of components and the integrated system because many existing application domains (multimedia, critical systems, and so forth) depend not only on correct functionality but also on how well it is achieved. UniFrame provides a mechanism for the component provider to specify the QoS parameters that are applicable to a provided component and determine the ranges that the component can guarantee.

Table 1 shows the UMM type specification of a component, *Validation Server*, for validating user accesses within the application domain of document management. In the advertised description of a corresponding implementation, the component provider would supply the actual values for various fields (such as N/A in Table 1). For example, the specification of a component that implements *Validation Server* would contain details, such as the URL where the component is deployed (id), the guaranteed values for the *throughput* and *end-to-end delay*, and the required deployment environment. The

Table 1. UMM type specification of a component

Abstract Component Type: *ValidationServer*

1. Component Name: *ValidationServer*
2. Domain Name: Document Management
3. System Name: DocumentManager
4. Informal Description: Provide the user validation service.
5. Computational Attributes:
    - 5.1 Inherent Attributes:
        - 5.1.1 id: N/A
        - 5.1.2 Version: version 1.0
        - 5.1.3 Author: N/A
        - 5.1.4 Date: N/A
        - 5.1.5 Validity: N/A
        - 5.1.6 Atomicity: Yes
        - 5.1.7 Registration: N/A
        - 5.1.8 Model: N/A
    - 5.2 Functional Attributes:
        - 5.2.1 Function description: Act as validation server for users in the system.
        - 5.2.2 Algorithm: N/A
        - 5.2.3 Complexity: N/A
        - 5.2.4 Syntactic Contract
        - 5.2.4.1 Provided Interface: *IValidation*
        - 5.2.4.2 Required Interface: NONE
        - 5.2.5 Technology: N/A
        - 5.2.6 Expected Resources: N/A
        - 5.2.7 Design Patterns: NONE
        - 5.2.8 Known Usage: Validation of user access
        - 5.2.9 Alias: NONE
6. Cooperation Attributes:
    - 6.1 Preprocessing Collaborators: *Users'Terminal*
    - 6.2 Postprocessing Collaborators: NONE
7. Auxiliary Attributes:
    - 7.1 Mobility: No
    - 7.2 Security: *L0*
    - 7.3 Fault tolerance: *L0*
8. Quality of Service Attributes
    - 8.1 QoS Metrics: *throughput, end-to-end delay*
    - 8.2 QoS Level: N/A
    - 8.3 Cost: N/A
    - 8.4 Quality Level: N/A
    - 8.5 Effect of Environment: N/A
    - 8.6 Effect of Usage Pattern: N/A
9. Deployment Attributes: N/A

specification associated with each implemented component is published when it is deployed on the network. The UMM specification of a component enhances the concept of a multilevel contract for components proposed by Beugnard, Jezequel, Plouzeau, and Watkins (1999) because it includes other details, such as bookkeeping, collaborative, algorithmic and technological information, and possible levels of service with associated costs and effects of different environmental factors on the QoS parameters.

Infrastructure

UniFrame assumes the presence of a publicly accepted knowledgebase that contains information, such as the component types needed for a specific application domain, the interconnections and constraints that make up the design specification of each component system in a domain, and rules for QoS calculations. Experts, such as standards organizations' task forces, create the UMM specifications for the components of each application domain of the knowledgebase. The UMM specifications of the component types are publicly distributed so that component developers can supply implementations that adhere to them.

UniFrame's Infrastructure consists of the System Generation Process, Resource Discovery Service (URDS), and Glue and Wrapper Generator. The first employs the knowledgebase to carry out the steps in creating a component system. It invokes the URDS to locate the components in the network the system requires and validates the product using an iterative process. The URDS provides mechanisms for components to publish their UMM specifications and for hosting the services on distributed machines, receives appropriate queries for locating the deployed services, and performs the selection of necessary components based upon specified criteria. It invokes the Glue and Wrapper Generator, which accommodates the heterogeneity across components, incorporates the mechanisms necessary to measure the QoS, and configures the selected services. Subsequent sections will provide more details about these.

Service-Oriented Architecture

In order to provide flexible, efficient support to the process of creating a DCS, UniFrame organizes its knowledgebase according to the concepts of Model-Driven Architecture proposed by the Object Management Group (2002) and Business Line Architecture proposed by the Enterprise Architecture SIG (2003a). UniFrame's UMM provides an underlying framework for this organization. The domain elements in the top tier of the architecture correspond to different business contexts, or lines. A context consists of a class of related business practice domains (such as, retail grocery, retail hardware, construction supply, wholesaler), which are located in the next tier down. Conceptually, elements on one level can share an element on another (health care and construction can share inventory), which differs in how it performs similar operations in different contexts (that is, the element comprises a set of variants). The various, hierarchically organized elements that contribute detail to the definition of a business context constitute its Business Reference Model, discussed in Succeeding with Component-Based Architecture by the Enterprise Architecture SIG (2003b). This takes the form of a tree, whose root represents the context in the architecture under consideration. Business domain experts perform requirements analysis and model the business contexts for which it is desired to construct DCSs. The Business Reference Models they derive and place in the knowledgebase define the space of problems UniFrame can solve.

For each Business Reference Model, software engineers construct design models in various ways to implement DCSs that satisfy its requirements. A design model is expressed, frequently in Unified Modeling Language (UML®) (Rumbaugh, Jacobson & Booch, 1999), in terms of tiered layers of components, each component offering a defined set of services. Several Business Reference Models can share components. A component in one tier can be composed (or use of the services) of components on a lower tier. Thus, a component has two definition forms in the knowledgebase:

- a specification of its abstract properties as a type, as in Table 1, or
- a design specification, following UMM standards, which directly references the components and refined design specifications that it uses.

The former is called an *abstract component*, which the UniFrame System Generation Process considers to be available with no construction necessary. The second form is called a *compound component*. The process will attempt to construct it from its design. A design specification that defines a realization of a Business Reference Model forms a Service Reference Model for it. It provides a vehicle for realizing the Model-Driven Architecture's mapping from a platform-independent model to a platform-specific model. The Service Reference Models also form part of UniFrame's knowledgebase.

In order to construct DCS solutions for a significant space of problems, the knowledgebase must contain matching (Business Reference Model, Service Reference Model) pairs for each problem variation anticipated. These can be organized efficiently by structuring related Business Reference Models in feature models according to the optional features that they exhibit and related Service Reference Models according to variation point archetypes that show which design variants are available. The experts create a domain-specific language based on the distinguishing features and variation points in the models. Then, users of the System Generation Process employ the language to specify their requirements. The following example illustrates the knowledgebase's organization.

Case Study

Suppose domain experts want to create a knowledgebase that includes the business context consisting of users who manage documents. The users' contact with the supporting system is via the use case *Manage Documents*, which includes *Validate User*. The use cases *Create Document, Delete Document, List Documents, Store Document,* and *Get Document* all extend *Manage Documents*. The last in this list includes *Lock Document*, whereas the others include *Unlock Document*. From the requirements these express, the domain experts identify three subsystems comprising the system: one for user validation, one for managing the documents themselves, and one for user interaction. The experts write a domain model for this system containing these three subsystems.

Suppose the experts decide the users may want to choose between two types of document manager systems: a standard document manager and a deluxe one that provides extended persistence support. They represent these options in a simplified feature diagram for the document manager, as shown in Figure 1. Clear small circles indicate optional features, whereas an arc indicates an exclusive OR choice. In more general feature diagrams (Griss, 2001), options of a node can be chosen as any combination of elements of a subset of the node's children. A feature diagram carries no information about how its alternatives might be associated with elements in the domain model of their parent node. It is an efficient mechanism for representing alternatives; the domain models are essential for representing the associations among elements in the models and the constraints on them. The domain model for the standard document manager consists of only one domain element, *Document Server*. The domain model for the deluxe document manager consists of two domain elements, *Deluxe Document Server* and its associated *Document Database* for persistence. Because there are just two alternatives in the feature diagram, there are just two Business Reference Models in this example. More generally, there will be as many as there are combinations permitted by the various feature diagrams present in the knowledgebase.

Software engineers experienced in the domain of the business context (document management here) develop design models for these two Business Reference Models. They create a service-oriented architecture of abstract components so that domain models map to component-based design models. Figure 2 shows the Service Reference Model, *Standard Document System*, for the Business Reference Model of the *Standard Document Manager* for this example. The Service Reference Model, *Deluxe Document System*, for the *Deluxe Document Manager* is identical, with the addition of a *Database* component associated with the *Document Server*, where the cardinality allows an arbitrary, positive number of *Database* units to be present. The Service Reference Models include the details defining the associations among the components. These might be views consisting of UML collaboration diagrams. This information is used to determine the entries in the UMM abstract component specifications and the interrelations of the components' interfaces. The specification for the abstract component, *Validation Server*, appeared in Table 1.

Suppose that the software engineers decide that two implementations of the standard document manager are possible, one in which the components adhere to .NET and the

Figure 1. Feature diagram for the document management system

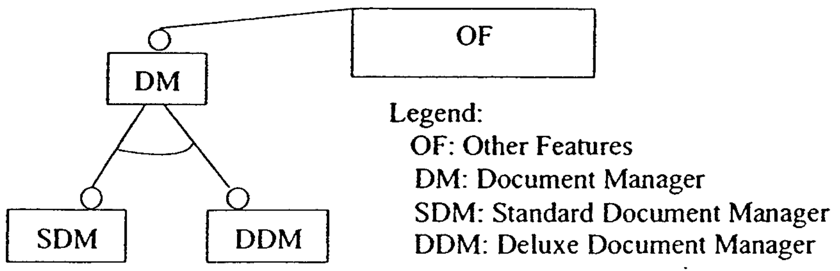
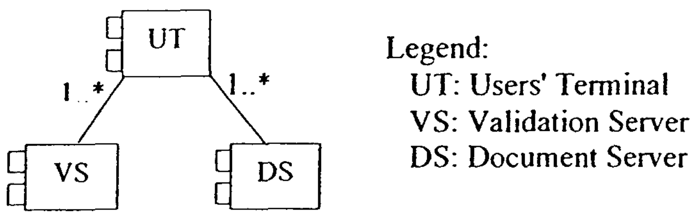


Figure 2. Service reference model for the standard document system



other to CORBA. They indicate this choice by a design model, labeled *Standard Document System*, augmented by variation point information that specifies the choice of one of these two technologies for the associations in Figure 2, such as in OCL (Warmer & Kleppe, 2003), as shown:

context Standard Document System

inv: technology = '.NET' or technology = 'CORBA'

Because the system consists of more than two components, the engineers have other combinations possible. For example, the *Users' Terminal/Validation Server* association may be in .NET technology, and the *Document Server* may be in CORBA technology, implying the need for an appropriate bridge.

UniFrame System Generation Process

The essential steps in UniFrame's process of constructing a DCS to solve a problem appear in Table 2. Once the UniFrame knowledgebase is available, a system developer can pose a statement of requirements for a DCS that solves a problem within its application domain. This analysis task forms step (1) in Table 2. For the case study in the previous section, the statement of requirements might be:

Create a Document Management System having a Standard Document Manager.

In step (2), the term *Document Management System* of the example requirements statement identifies the business context, so the stated problem lies within the domain the knowledgebase represents. The corresponding system model shows there are two alternatives for the *Document Manager*, which the feature model displays in Figure 1. The qualifying requirement, *Standard*, resolves this ambiguity, which completes step (2). The resulting Business Reference Model maps directly in the knowledgebase to the two alternative platform-specific Service Reference Models for the entire system shown in

Table 2. Steps in the UniFrame System Generation Process

Steps	Activities
1	State the requirements the DCS must satisfy in the knowledgebase's terminology.
2	Identify a Business Reference Model that represents these.
3	Identify each Service Reference Model specifying a system of abstract components that satisfies the Business Reference Model.
4	Obtain concrete implementations of the abstract components.
5	Assemble the concrete components into a DCS according to each Service Reference Model, so that it meets the specified requirements.
6	Test the DCS against the requirements and exit if satisfactory; otherwise, return to step (1) to modify the requirements.

Figure 2, in which the components are either all .NET or all CORBA. This completes step (3).

Continuing to step (4), the System Generation Process collects the UMM type specifications of all the abstract components involved in each of the two Service Reference Models and sends them in a query to the UniFrame Resource Discovery Service. This searches the network for implemented components whose UMM descriptions satisfy the type specifications.

Step (5) employs the design information in a Service Reference Model to construct a DCS with the components found. If the appropriate implementations are available on the network, the request for a *Standard Document Manager* in the example will yield two DCSs, one with .NET technology and one with CORBA technology. If no .NET implementation of a *Validation Server* is found, then only the CORBA DCS will be constructed.

Typically, a developer understands the requirements poorly at the initiation of the System Generation Process. Therefore, it is imperative to evaluate empirically the consistency of the characteristics of a generated DCS with the perceived requirements and make modifications as necessary. This motivates having step (6) in Table 2. Such iterative development provides a mechanism for the developer to validate the outcome of the process and determine empirically the ranges within which its QoS attributes vary. This helps to assure a higher quality product. The process allows two levels of testing. The simplest is black box (or acceptance) testing of the DCS based on only the stated requirements. The developer supplies a test harness and plan for this. The other is white box (or integration) testing, again based on the developer's test plan. In this case, the design of the DCS serves as a guide for inserting instrumentation code between the components in the DCS. At runtime, this code reports the behavior of the DCS, giving the developer a view into its internal operation. The section on the measurement of QoS discusses a mechanism for inserting this instrumentation easily.

In case there are several Business or Service Reference Models in the knowledgebase that satisfy the developer's requirements if step (2) or (3) of the process provides feedback, allowing the developer to introduce requirements incrementally so as to reduce these alternatives, then the process becomes an efficient way to construct the needed type of DCS. Thus, the System Generation Process supports the iterative, incremental development paradigm that modern software engineering practices have found productive.

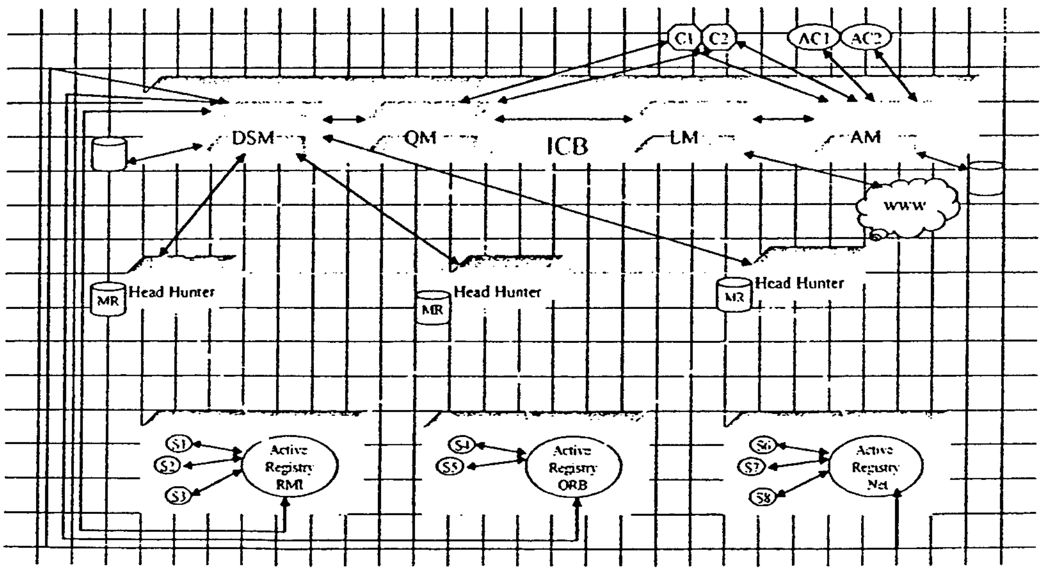
UniFrame Resource Discovery Service (URDS)

Once components and their UMM descriptions have been deployed on the network, they are ready for discovery in the UniFrame System Generation Process. The URDS executes this process. Siram et al. (2002) discuss its architecture, shown in Figure 3.

The URDS architecture comprises: HeadHunters (HHs), Internet Component Broker (ICB), Meta-repositories (MRs), and components.

Components are implemented according to some component model, as described earlier, and registered with the model's binding service. For example, the Java-RMI components are registered with the Naming service provided by the Java-RMI framework. An

Figure 3. UniFrame Resource Discovery System (URDS)



advantage of this is that it does not burden the component providers because, to deploy their implementations, they must register them anyway. The HHs have the sole responsibility of performing matchmaking operations between registered components and requested specifications. Each HH has an MR, which serves as a local store. An HH is constantly discovering newly implemented components and storing their UMM specifications in its MR. Anytime an HH receives a query for a component type, it first searches its MR. If it finds a match, it returns the corresponding component as a result. If not, it propagates the query to other HHs in the system.

The ICB is analogous to the object request broker (ORB) in other architectures. Unlike the ORB, which only allows interoperation between components having heterogeneous implementations, the Internet component broker allows interoperation between components with different component models. As Figure 3 shows, the Internet component broker consists of domain security manager (DSM), query manager (QM), link manager (LM), and adapter manager (AM). The DSM is responsible for enforcing a security structure on the URDS. It authenticates the HHs and allows them to communicate with different binding mechanisms (registries). The QM interfaces with the System Generation Process. It receives a query consisting of a collection of UMM component types, passes it to the HHs, and returns the results. The LM allows a federation of URDSs to be created in order to increase the component search space. The AM locates adapter components, such as bridges that allow interoperation of different component models, and passes them to the Glue and Wrapper Generator.

A prototype of URDS has been implemented using the Java-RMI and .NET technologies. Many experiments have been performed to measure its performance (Siram et al., 2002). These demonstrate that URDS scales upward, but the details extend beyond this chapter's scope.

Industry and academia have proposed and implemented many distributed resource discovery and directory services. Examples that Siram et al. (2002) describe include WAIS, Archie, Gopher, UDDI, CORBA Trader, LDAP, Jini, SLP, Ninf, and NetSolve. Each has its own characteristics and exhibits some similarity with URDS. The distinguishing features of URDS are its treatment of heterogeneity and its purpose to support creating heterogeneous integrated systems, not just to discover services.

UniFrame Quality of Service Framework (UQoS)

Components offer services and indicate and guarantee the quality of their services. Therefore, it is necessary to facilitate the publication, selection, measurement, and validation of component and DCS QoS values. The UniFrame Quality of Service Framework, described by Brahnmath (2002); Sun (2003); and Raje, Bryant, Olson, Auguston, and Burt (2002), provides necessary guidelines for the component developers and system integrators using UniFrame. The UQoS consists of three parts: QoS catalog, composition/decomposition models for QoS parameters, and specification and measurement of QoS. The reader is referred to the references above for the first two because the details are extensive.

To prepare the UMM description of a component to be publicized, the component developer must measure empirically the QoS parameters in the corresponding UMM type specification. The QoS catalog provides model definitions and formulas to assist in this. Some parameters are static in nature (like reliability), while some are dynamic (like end-to-end delay). If the parameter is static and characterizes a system of components, then its value can be determined from the components' parameter values. Otherwise, its value must be determined empirically.

Evaluation of QoS Parameters

UniFrame uses the principles of event grammars for measuring parameters empirically. Event grammar, as described by Auguston (1995), forms the basis for system behavior models. An event represents any detectable action during execution, such as a statement execution, expression evaluation, procedure call, and receiving a message. It has a beginning, end, and duration (a time interval corresponding to the action of interest). Actions (or events) evolve in time, and system behavior represents the temporal relationship among actions. This implies a partial ordering relation for events, as Lamport (1978) discussed.

System execution can be modeled as a set of events (event trace) with two basic relations: partial ordering and inclusion. The event trace actually is a model of the system's temporal behavior. In order to specify meaningful system behavior properties, events must be enriched with attributes. An event may have a type and other attributes, such as duration, source code related to the event, associated state (that is, variable values at the event's beginning and end), and function name and returned value for function call events.

A special programming language, FORMAN, for computations over event traces greatly facilitates measuring parameters empirically. As described by Fritzson, Auguston, and Shahmehri (1994) and Auguston (1995), it is based on the notions of the functional paradigm, event patterns, and aggregate operations over events.

The execution model of a component (or a system of integrated components) is defined by an event grammar, which is a set of axioms that describes possible patterns of basic relations between events of different types in a program execution trace. It is not intended to be used for parsing actual event traces. If an event is compound, the grammar describes how it splits into other event sequences or sets. For example, the event *execute-assignment-statement* contains a sequence of events *evaluate-right-hand-part* and *execute-destination*.

The rule $A :: (B\ C)$ establishes that, if an event a of the type A occurs in the trace of a program, it is necessary that events b and c of types B and C, also exist, such that the relations b IN a, c IN a, b PRECEDES c hold. For example, the event grammar describing the semantics of an imperative programming language may contain the following rule (the names, such as *execute-program* and *ex-stmt* in the grammar denote event types):

*execute-program :: (ex-stmt *)*

This means that each event of the type *execute-program* contains an ordered (w.r.t. relation PRECEDES) sequence of zero or more events of the type *ex-stmt*. For the function call event, the event grammar may provide the following rule:

*func_call:: (param *) (ex-stmt *)*

This event may contain zero or more parameter evaluation events followed by statement executions.

Example of Evaluating Turn-Around Time

If the event type *component_call* corresponds to the whole component call event and *request* denotes the event for a single request (the time interval from the request's beginning to its completion), then the following FORMAN formula specifies the measurement of the turn-around time:

```
FOREACH a: session FROM execute_program
SAY ( 'Turn-around Time for a session is '
    SUM[ b: request FROM a APPLY b.duration]
                         / CARD[ request FROM a] )
```

Similar rules can be specified for any other dynamic QoS parameters or related computations. Thus, the principles of event traces provide a mechanism to validate empirically the QoS values for a component and for an integrated system of components.

Interoperability Using the Glue and Wrapper Generator

For interoperation of heterogeneous distributed components, it is necessary to construct glue and wrapper code to interconnect the components. Because a project objective is to achieve high quality systems, a goal is to automatically generate the glue/wrapper code. In order to achieve this, there should be formal rules for interconnecting components from a specific application domain as well as integration of multiple technology domains, that is, component models. UniFrame uses the Two-Level Grammar (TLG, also called W-grammar) formal specification language (Bryant & Lee, 2002) to specify both types of rules. The TLG formalism is used to specify the components deployed under UniFrame and also the generative rules needed for system assembly. The output of the TLG will provide the desired target code (for example, glue and wrappers for components and necessary infrastructure for the distributed runtime architecture). The UMM formalization establishes the context for which the generative rules may be applied. Bryant, Auguston, Raje, Burt, and Olson (2002) provide further details about the glue/wrapper code generation rules, including a discussion of how the Quality of Service validation code is inserted into the glue code. The general principle is that for each QoS parameter to be dynamically verified, the glue code is instrumented according to the event grammar rules described earlier.

Future Trends

The concept of Business Reference Models "is meant to provide the foundation for common understanding of business processes across the Federal government in a service-oriented manner," enabling an agency to define an enterprise architecture as mandated by law (Enterprise Architecture SIG, 2003). A significant sector of industry is involved in establishing standards and guidelines on how to enable successful enterprise architecture. The component-based architecture of UniFrame's knowledgebase closely follows these guidelines, incorporating the concepts of Object Management Group's (2002) Model-Driven Architecture as an integral part. Consequently, UniFrame is working toward the realization of an operational framework for enterprise architecture and is a source of feedback into the activities necessary.

Many existing component models provide the necessary mechanisms for describing the functional aspects of components but not for the QoS aspects. Standards organizations have recently started to address this weakness. For example, in the fall of 2000, the OMG began issuing a number of Requests for Proposals for UML profiles for modeling QoS in several contexts. UniFrame is addressing some of these QoS issues and is making efforts (via presentations to different OMG task forces) to ensure that its research is aligned with emerging industry standards.

The creation of the Business Line and Service-Oriented knowledgebase will largely continue to be a human endeavor aided by CASE tools because humans determine what constitutes the problems they must solve. However, the System Generation Process could be accomplished mostly automatically for any problem in a given knowledgebase. The person who formulates the requirements for the DCS will need to do so in the knowledgebase's terminology. The degree to which this can be made to match the typical user's terminology remains a research area.

Huang (2003) implemented a prototype of the UniFrame System Generation Process with the UniFrame Resource Discovery Service. Because of the labor involved in constructing the knowledgebase, it was limited to a small banking case study. Experimental studies proved efficient, user communication issues were easily managed, and QoS values were calculated. The automated creation of bridges and glue/wrapper code and using FORMAN to insert the code into them for the QoS computations remain to be incorporated into the implementation.

Conclusion

This chapter has described the UniFrame process for constructing distributed computing systems and has shown how it facilitates achieving the current goals of government and industry in rapidly creating high quality computing systems. UniFrame provides a framework within which a diverse array of technologies can be brought to achieve these ends. These include software engineering practices, such as rapid, iterative, and incremental development. Its business line, service-oriented, model-driven architecture based on components is a realization of the movement to provide mutability, quick development, and conservation of resources. A knowledgebase of component-based, predefined and tested designs for distributed computing systems, event traces for empirical testing, and quality of service prediction and calculation are tools it utilizes for increasing quality assurance. UniFrame decouples the requirements analysis and system assembly activities from the problem of collecting appropriate components published on the network. Its novel resource discovery service facilitates the efficient acquisition of components meeting stated specifications. It provides a mechanism for seamlessly bridging components of different models, such as RMI and CORBA, to support the construction of heterogeneous, distributed computing systems having platform-independent definitions. The UniFrame project is also investigating techniques and patterns related to using quality of service parameters during the design of components and integrated systems to create high assurance distributed computing systems.

Acknowledgments

This work was supported in part by the U.S. Office of Naval Research, grant N00014-01-1-0746.

References

Auguston, M. (1995). *Program behavior model based on event grammar and its application for debugging automaton.* In M. Ducassé (Ed.), Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95) (pp. 277-291), Rennes: Université de Rennes.

Batory, D., & Geraci, B. (1997). Component validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering, 23*(2), 67-82.

Beugnard, A., Jezequel, J., Plouzeau, N., & Watkins, D. (1999). Making components contract aware. *IEEE Computer, 32*(7), 38-45.

Brahnmath, G. (2002). *The UniFrame Quality of Service Framework.* Unpublished master's thesis, Indiana University Purdue University, Indianapolis, IN, United States. Retrieved August 8, 2004: *http://www.cs.iupui.edu/uniFrame/*

Brown, A. (1999). Building systems from pieces with component-based software engineering. In P. Clements (Ed.), *Constructing superior software* (Chapter 6). Indianapolis, IN: MacMillan Technical.

Bryant, B. R., Auguston, M., Raje, R. R., Burt, C. C., & Olson, A. M. (2002). *Formal specification of generative component assembly using two-level grammar.* Proceedings of SEKE 2002, 14th International Conference on Software Engineering and Knowledge Engineering (pp. 209-212). Los Alamitos: IEEE Press.

Bryant, B. R., & Lee, B.-S. (2002). *Two-Level grammar as an object-oriented requirements specification language.* Proceedings of HICSS-35, the 35th Hawaii International Conference on System Sciences (p. 280). Los Alamitos, CA: IEEE Press. Retrieved August 8, 2004: *http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf*

Crnkovic, I., Schmidt, H., Stafford, J., & Wallnau, K. (Eds.). (2003). Proceedings of the 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction. The 25th International Conference on Software Engineering (ICSE). Retrieved August 8, 2004: *http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6*

Enterprise Architecture SIG, Industrial Advisor Council (IAC). (2003a, March). Business line architecture and integration. Retrieved August 8, 2004: *http://216.219.201.97/documents_presentations/index.htm*

Enterprise Architecture SIG, Industrial Advisor Council. (2003b, March). (IAC). Succeeding with component-based architecture in e-government. Retrieved August 8, 2004: *http://216.219.201.97/documents_presentations/index.htm*

Fritzson, P., Auguston, M., & Shahmehri, N. (1994). Using assertions in declarative and operational models for automated debugging. *The Journal of Systems and Software, 25,* 223-239.

Griss, M. L. (2001). Product line architectures. In G. T. Heineman, & W. T. Councill (Eds.), *Component-based software engineering: Putting the pieces together* (pp. 405-420). Boston: Addison-Wesley.

Heineman, G. T., & Councill, W. T. (Eds.). (2001). *Component-based software engineering: Putting the pieces together.* Boston: Addison-Wesley.

Huang, Z. (2003). *The UniFrame system-level generative programming framework.* Unpublished master's thesis, Indiana University Purdue University, Indianapolis, IN, United States. Retrieved August 8, 2004: *http://www.cs.iupui.edu/uniFrame*

Java Remote Method Invocation – Distributed computing for Java. (2003, October 2). Retrieved August 8, 2004: *http://java.sun.com/marketing/collateral/javarmi.html*

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM, 21*(7), 558-565.

Lumpe, M., Schneider, J., Nierstrasz, O., & Achermann, F. (1997). *Towards a formal composition language.* In G. T. Leavens & M. Sitamaran (Eds.), Proceedings of the 1st ESEC Workshop on Foundations of Component-Based Systems (pp. 178-187). Heidelberg: Springer-Verlag.

Microsoft .Net Framework: Technology overview. (2003, October 2). Retrieved August 8, 2004: *http://msdn.microsoft.com/netframework/technologyinfo/overview/*

Object Management Group. Model-Driven Architecture™, the architecture of choice for a changing world. (2002, March 12). Retrieved August 8, 2004: *http://www.omg.org/mda*

Raje, R. (2000). *UMM: Unified Meta-object Model for open distributed systems.* Proceedings of the Fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP 2000) (pp. 454-465). Los Alamitos, CA: IEEE Press.

Raje, R., Auguston, M., Bryant, B., Olson, A., & Burt, C. (2001). *A unified approach for integration of distributed heterogeneous software components.* Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration, SEAC technical report (pp. 109-119). Monterey, CA: U.S. Naval Postgraduate School. Retrieved August 8, 2004: *http://www.cs.iupui.edu/uniFrame/*

Raje, R., Bryant, B., Olson, A., Auguston, M., & Burt, C. (2002). A quality-of-service-based framework for creating distributed heterogeneous software components. *Concurrency and Computation: Practice and Experience, 14,* 1009-1034.

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language reference manual.* Reading, MA: Addison Wesley.

Schmidt, D. (2003, October 2). Overview of CORBA. Retrieved August 8, 2004: *http://www.cs.wustl.edu/~schmidt/corba-overview.html*

Siram, N., Raje, R., Olson, A., Bryant, B., Burt, C., & Auguston, M. (2002). *An architecture for the UniFrame Resource Discovery Service.* Proceedings of the 3rd International Workshop of Software Engineering and Middleware: Vol. 2596. Lecture Notes in Computer Science (pp. 20-35). Heidelberg: Springer-Verlag.

Sun, C. (2003). *QoS composition and decomposition models in UniFrame.* Unpublished master's thesis, Indiana University Purdue University, Indianapolis, IN, United States. Retrieved August 8, 2004: *www.cs.iupui.edu/uniFrame*

Szyperski, C., Gruntz, D., & Murer, S. (2002). *Component software - Beyond object-oriented programming.* (2nd ed.). Boston: Addison-Wesley/ACM Press.

Warmer, J., & Kleppe, A. (2003). *The Object Constraint Language.* (2nd ed.). Boston: Addison-Wesley.

Weck, W. (1997, June). Independently extensible component frameworks. In M. Mühlhäuser (Ed.), *Proceedings of the 1st International Workshop on Component-*

Oriented Programming (European Conference on Object-Oriented Programming, Jyväskylä, Finland), *Special Issues in Object-Oriented Programming* (pp. 177-188). Heidelberg: Springer-Verlag.